A Simple Concurrent Lambda Calculus for Session Types

Jules Jacobs

Radboud University Nijmegen, mail@julesjacobs.com

Abstract

We introduce \( \mu \text{GV} \) (“micro GV”), which strives to be a minimal extension of linear \( \lambda \)-calculus with concurrent communication, adding only a new \texttt{fork} construct for spawning threads. The child and parent thread communicate with each other via two dual values of linear function type \( \tau_1 \rightarrow^\text{lin} \tau_2 \) and \( \tau_2 \rightarrow^\text{lin} \tau_1 \). Using only \texttt{fork}, we can implement all of GV’s channel operations and session types as a library in \( \mu \text{GV} \). The linear type system ensures that \( \mu \text{GV} \) 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 \( \lambda \)-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 \( \lambda \)-calculus, one adds session type formers and their corresponding channel operations: \( !\tau.s \) (send a message of type \( \tau \), continue with protocol \( s \)), \( ?\tau.s \) (receive a message of type \( \tau \), 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 \texttt{End} (close channel). An example is \( !\tau_1.(?\tau_2.\texttt{End} \oplus !\tau_3.\texttt{End}) \): send a value of type \( \tau_1 \) then either receive \( \tau_2 \) or send \( \tau_3 \). One adds \texttt{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 \texttt{!} dual to \texttt{?}, \( \oplus \) dual to \( \& \), and \texttt{End} dual to itself.

\( \mu \text{GV} \), on the other hand, has none of these. Instead, we add only a single construct: \texttt{fork}.

\[
\texttt{fork} : ((\tau_1 \rightarrow^\text{lin} \tau_2) \rightarrow (\tau_2 \rightarrow^\text{lin} \tau_1))
\]

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

\[
\texttt{let } c = (\texttt{fork}(\lambda c'. \texttt{print}(c' \ 1))) \ \texttt{in } \texttt{print}(1 + c \ 0)
\]

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 \( c' \) is called, it will block until \( c \) is also called, and vice versa. The call \( c' \) will then return \( 0 \), and the call \( c \) 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:

\[
\begin{align*}
\texttt{send}(c, x) &\triangleq \texttt{fork}(\lambda c'. \ c(c', x)) & \texttt{receive}(c) &\triangleq c() & \texttt{close}(c) &\triangleq c()
\end{align*}
\]

\[
\begin{align*}
\texttt{let } x_1 &= \texttt{fork}(\lambda x'. \ c(x')) & \texttt{receive}(c) &= c() & \texttt{close}(c) &= c()
\end{align*}
\]

\[
\begin{align*}
\texttt{let } x_1 &= \texttt{fork}(\lambda x'. \ c(x')) & \texttt{receive}(c) &= c() & \texttt{close}(c) &= c()
\end{align*}
\]

\[
\begin{align*}
\texttt{let } x_2 &= \texttt{send}(x_1, 1) & \texttt{send message } 1
\end{align*}
\]

\[
\begin{align*}
\texttt{let } x_3 &= \texttt{send}(x_2, 2) & \texttt{send message } 2
\end{align*}
\]

\[
\begin{align*}
\texttt{let } x_4 &= \texttt{receive}(x_3) & \texttt{receive } n &= 1 + 2
\end{align*}
\]

\[
\begin{align*}
\texttt{print}(n) &\texttt{ close}(x_4)
\end{align*}
\]
A Simple Concurrent Lambda Calculus for Encoding Session types

Jules Jacobs

$\mu$GV expressions and types

\begin{align*}
e \in \text{Expr} & \coloneqq \ x \mid () \mid (e,e) \mid \text{in}_L(e) \mid \text{in}_R(e) \mid \lambda x. \ e \mid e \ e \mid \text{fork}(e) \\
\text{let } (x_1, x_2) & = e \text{ in } e \mid \text{match } e \text{ with } \ldots \text{ end}
\end{align*}

$\tau \in \text{Type} \coloneqq 0 \mid 1 \mid \tau \times \tau \mid \tau \oplus \tau \mid \tau \text{ lin} \rightarrow \tau \mid \tau \text{ unr} \rightarrow \tau \mid \mu x. \tau \mid x$

Session types duality

\begin{align*}
\text{End} & \triangleq \text{End} \\
!\tau.s & \triangleq ?\tau.\overline{s} \\
?\tau.s & \triangleq !\tau.\overline{s} \\
\overline{s_1} \oplus \overline{s_2} & \triangleq \overline{s_1 \land s_2} \\
\overline{s_1} \land \overline{s_2} & \triangleq \overline{s_1 \land s_2}
\end{align*}

Encoding session types in $\mu$GV

\begin{align*}
\text{End} & \triangleq 1 \text{ lin} \rightarrow 1 \\
!\tau.s & \triangleq s \times \tau \text{ lin} \rightarrow 1 \\
?\tau.s & \triangleq 1 \text{ lin} \rightarrow s \times \tau \\
\overline{s_1} \oplus \overline{s_2} & \triangleq s_1 + s_2 \text{ lin} \rightarrow 1 \\
\overline{s_1} \land \overline{s_2} & \triangleq \text{ lin} \rightarrow s_1 + s_2
\end{align*}

Channel operations

<table>
<thead>
<tr>
<th>Channel operations</th>
<th>Encoding channel operations in $\mu$GV</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork : $(s \text{ lin} \rightarrow 1) \rightarrow \overline{s}$</td>
<td>fork$(x) \triangleq \text{fork}(x)$</td>
</tr>
<tr>
<td>close : End $\rightarrow$ 1</td>
<td>close$(c) \triangleq c()$</td>
</tr>
<tr>
<td>send : $!\tau.s \times \tau \rightarrow s$</td>
<td>send$(c, x) \triangleq \text{fork}(\lambda c'. c (c', x))$</td>
</tr>
<tr>
<td>receive : $?\tau.s \rightarrow s \times \tau$</td>
<td>receive$(c) \triangleq c()$</td>
</tr>
<tr>
<td>tell$_L : s_1 \oplus s_2 \rightarrow s_1$</td>
<td>tell$_L(c) \triangleq \text{fork}(\lambda c'. c \text{ in}_L(c'))$</td>
</tr>
<tr>
<td>tell$_R : s_1 \oplus s_2 \rightarrow s_2$</td>
<td>tell$_R(c) \triangleq \text{fork}(\lambda c'. c \text{ in}_R(c'))$</td>
</tr>
<tr>
<td>ask : $s_1 \land s_2 \rightarrow s_1 + s_2$</td>
<td>ask$(c) \triangleq c()$</td>
</tr>
</tbody>
</table>

Figure 1: The $\mu$GV language (top), session types (left) and their encoding in $\mu$GV (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 $\mu$GV’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 $\mu$GV types.

Like GV, all well-typed $\mu$GV programs are automatically deadlock free, and therefore satisfy global progress. We prove this property in Coq. Because of $\mu$GV’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 $\mu$GV (1442 lines vs 2796 lines).

There have been other efforts for simpler systems, such as an encoding of session types into $\pi$-calculus types [2], and minimal session types [1], which decompose multi-step session types into single-step session types in a $\pi$-calculus (single-shot synchronization primitives have also been used in the implementation of a session-typed channel library for Haskell [9]).

I hope that $\mu$GV shows that linear $\lambda$-calculus also provides a good substrate for a minimalist concurrent calculus, with communication primitives that capitalize on the fact that the quintessential linear $\lambda$-calculus type, the linear function type $\tau_1 \text{ lin} \rightarrow \tau_2$, is a self-dual connective.
References