2019 Korea-EU Researcher Exchange Program

A Theory of RPC Calculi for Client-Server Model

Kwanghoon Choi

Dept. of Electronics and Computer Engineering
Chonnam National University
Republic of Korea

July 2019

(Cowork with Byeong-Mo Chang at Sookmyung Women’s University, Korea)
Tierless programming languages for client-server model such as Web to address the client-server dichotomy

- To allow to intersperse client and server expressions with seamless communication in a unique PL
- To support an automatic slicing of the program into two parts which run on the server and on the client, respectively
Introduction

The RPC calculus, the simplest semantics foundation for the tierless programming languages (Cooper&Wadler, 2009)

▶ Uses the syntax of \(\lambda\)-application for remote procedure calls

\[
\text{main}^{c} = \text{authenticate} () \\
\text{authenticate} = \lambda^{s}x. \\
\quad \text{let creds = getCredentials "Enter name:passwd > " in} \\
\quad \quad \text{if creds == "ezra:opensesame"} \\
\quad \quad \quad \text{then "the secret document" else "Access denied"}
\]

\[
\text{getCredentials} = \lambda^{c}\text{prompt}. (\text{print promt}; \text{read})
\]
Problem

The other tierless calculi (ML5, Hop, Ur/Web, Eliom, etc.) support asymmetric communication, or they are for peer-to-peer model.

Only the RPC calculus supports symmetric communication programming for client-server model.

However, in the original (untyped) RPC calculus,

- The semantics foundation for the stateless server style, not for the stateful server style
- The complicate compilation rules due to the absence of location information in lambda applications
In this research

A theory of RPC calculi for client-server model

- A typed version of the RPC calculus that can account for remote procedure calls in type level
- Type-directed slicing compilations in the stateless style, the stateful style, and the mixed style
- Establishment of type soundness of the locative type system and the correctness of the compilations
Part I: A typed RPC calculus

A locative type system for the RPC calculus that identifies remote procedure calls statically

<table>
<thead>
<tr>
<th>RPC calculi</th>
<th>CS(Client-Server) calculi</th>
</tr>
</thead>
<tbody>
<tr>
<td>(single unified term)</td>
<td>(separate terms)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RPCs identified at type level</th>
<th>RPCs identified at term level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_{rpc}$</td>
<td>$\lambda_{cs}$</td>
</tr>
<tr>
<td>(locative type system)</td>
<td>(stateless server)</td>
</tr>
</tbody>
</table>

| $\lambda_{rpc}$                | $\lambda_{enc}$               |
| (stateless server)             | (stateful server)             |

| $\lambda_{state}$              | $\lambda_{state}$             |
| (stateful server)              | (stateful server)             |

**Diagram:**
- $\lambda_{rpc}$ (locative type system)
- $\lambda_{enc}$ (stateless server)
- $\lambda_{state}$ (stateful server)
The RPC calculus

A call-by-value $\lambda$-calculus with location annotated $\lambda$-abstractions

$\lambda^a x. N : \lambda$-abstraction that must run at location $a$

- Location $a, b ::= c \mid s$
- Term $L, M, N ::= V \mid L M$
- Value $V, W ::= x \mid \lambda^a x. N$

Evaluation

\[
\begin{align*}
V \downarrow_a V & \quad \text{(Value)} \\
L \downarrow_a \lambda^b x. N & \quad M \downarrow_a W & \quad N\{W/x\} \downarrow_b V \\
& \quad L M \downarrow_a V & \quad \text{(Beta)}
\end{align*}
\]
The RPC calculus

An example of symmetric communication flow between the client and the server:

\[(\lambda^s f. (\lambda^s x. x) (f c)) (\lambda^c y. (\lambda^s z. z) y)\]
A locative type system for the RPC calculus

Every $\lambda$-abstraction of type $\tau \xrightarrow{a} \tau'$ runs at location $a$.

Type $\tau ::= base \mid \tau \xrightarrow{a} \tau$

$\triangleright (\lambda^s f. (\lambda^s x.x) (f \ M)) (\lambda^c y. (\lambda^s z.z) \ y)$

Well-typed where $f : \tau_1 \xrightarrow{c} \tau_2$

$\triangleright (\lambda^c f. f \ M) (\text{if} \ \cdots \ \text{then} \ \lambda^c x. M_1 \ \text{else} \ \lambda^s y. M_2)$

Ill-typed because neither $f : \tau_1 \xrightarrow{c} \tau_2$ nor $f : \tau_1 \xrightarrow{s} \tau_2$
A locative type system for the RPC calculus

A typing judgment, $\Gamma \triangleright_a M : \tau$, says:

- A term $M$ at location $a$ has type $\tau$ under a type environment $\Gamma$

Key idea: A refinement of the lambda application typing w.r.t. the combinations of location $a$ and location $b$

\[
\frac{\Gamma \triangleright_a L : \tau \xrightarrow{b} \tau' \quad \Gamma \triangleright_a M : \tau}{\Gamma \triangleright_a L \ M : \tau'}
\]

cf. \[
L \Downarrow_a \lambda^b x. N \quad M \Downarrow_a W \quad N\{W/x\} \Downarrow_b V \quad \frac{L \ M \Downarrow_a V}{(Beta)}
\]
A locative type system for the RPC calculus

The use of (T-App), (T-Req), and (T-Call) says ‘L M’ is a local procedure call, a c-to-s RPC, and a s-to-c RPC, respectively.

(T-Var) \( \frac{\Gamma(x) = \tau}{\Gamma \triangleright_a x : \tau} \)

(T-Lam) \( \frac{\Gamma\{x : \tau\} \triangleright_b M : \tau'}{\Gamma \triangleright_a \lambda^b x. M : \tau \rightarrow \tau'} \)

(T-App) \( \frac{\Gamma \triangleright_a L : \tau \rightarrow \tau' \quad \Gamma \triangleright_a M : \tau}{\Gamma \triangleright_a L \ M : \tau'} \)

(T-Req) \( \frac{\Gamma \triangleright_c L : \tau \rightarrow_s \tau' \quad \Gamma \triangleright_c M : \tau}{\Gamma \triangleright_c L \ M : \tau'} \)

(T-Call) \( \frac{\Gamma \triangleright_s L : \tau \rightarrow_c \tau' \quad \Gamma \triangleright_s M : \tau}{\Gamma \triangleright_s L \ M : \tau'} \)
A locative type system for the RPC calculus

A example of a typing derivation where \( f : base \overset{c}{\rightarrow} base \),

\[
(\lambda^s f. \ (\lambda^s x.x)^{s_3} (f \overset{c_1}{\rightarrow} c))^{s_1} (\lambda^c y. \ (\lambda^s z.z)^{s_2} y)
\]

using the following notation:

- **Notation**: \( L \overset{b}{\rightarrow} M \) for

\[
\frac{\Gamma \overset{a}{\rightarrow} L : \tau \overset{b}{\rightarrow} \tau'}{\Gamma \overset{a}{\rightarrow} L \ M : \tau'}
\]

cf.

\[
(\lambda^s f. \ (\lambda^s x.x)^{s_3} (f \overset{c_1}{\rightarrow} c))^{s_1} (\lambda^c y. \ (\lambda^s z.z)^{s_2} y)
\]
Properties of the locative type system

Type soundness for the RPC calculus

- If $\Gamma \triangleright_a M : \tau$ and $M \Downarrow_a V$, then $\Gamma \triangleright_a V : \tau$.

Corollary: Every remote procedure call identified statically will never change to a local procedure call under evaluation.
Properties of the locative type system

Typeability for the RPC calculus

- Every simply typed term with arbitrary location annotations is (or can be transformed to be) typed under our type system.

- \((\lambda^a f. \ldots)\) \(M\) is ill-typed where \(f : \tau \xrightarrow{c} \tau', M : \tau \xrightarrow{s} \tau'\), but \((\lambda^a f. \ldots)\) \((\lambda^c x. M \ x)\) is well-typed where \((\lambda^c x. M \ x) : \tau \xrightarrow{c} \tau'\).

\[\text{\([M]\)}^{\tau_1 \leadsto \tau_2}: \text{Location transformation of a term } M \text{ of type } \tau_1 \text{ into } \tau_2\]

\[
\begin{align*}
\text{\([M]}^{\tau \leadsto \tau} & = M \\
\text{\([M]}^{\tau_1 \xrightarrow{a} \tau_2 \leadsto \tau_3 \xrightarrow{b} \tau_4} & = \lambda^b x.\text{\([M]}^{\tau_3 \leadsto \tau_1}\text{\([x]}^{\tau_2 \leadsto \tau_4}}
\end{align*}
\]
Part II: Slicing with state-encoding calculi

A server stateless implementation for scalability, leaving no states on the server after each cycle of request-response

\[ \lambda_{rpc} \Rightarrow \lambda_{rpc}^{enc} \Rightarrow \lambda_{cs}^{enc} \]

cf. (Cooper&Wadler, 2009)
Basic idea

Collapsing arbitrarily deep symmetric communication into a series of request-response leaving no states on the server
A state-encoding RPC calculus $\lambda^{enc}_{rpc}$

\[
\lambda_{rpc} \Rightarrow \lambda^{enc}_{rpc} \Rightarrow \lambda^{enc}_{cs}
\]

In $\lambda^{enc}_{rpc}$, remote procedure calls explicitly in term-level as:

\[
\text{Term } M ::= V \mid \text{let } x = M \text{ in } M \mid V_f(W) \mid \text{req}(V_f, W) \mid \text{call}(V_f, W)
\]

In the compilation of $\lambda_{rpc}$ into $\lambda^{enc}_{rpc}$,

- A typing derivation directed compilation

- Continuation-passing style (CPS) for encoding the rest of the server evaluation right after each client function call
Compilation of $\lambda_{rpc}$-typing derivations into $\lambda_{rpc}^{enc}$

Direct style compilation for the client part, and CPS compilation for the server part

**Client:**

$C[M_{rpc}] = M_{rpc}^{enc}$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compilant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C[x]$</td>
<td>$x$</td>
</tr>
<tr>
<td>$C[\lambda_{x}.M]$</td>
<td>$\lambda_{x}.C[M]$</td>
</tr>
<tr>
<td>$C[\lambda_{s}.M]$</td>
<td>$\lambda_{s}(x, k).S[M] k$</td>
</tr>
<tr>
<td>$C[L_{c}M]$</td>
<td>let $f = C[L]$ in let $x = C[M]$ in let $r = f(x)$ in $r$ cf. (T-App)</td>
</tr>
<tr>
<td>$C[L_{s}M]$</td>
<td>let $f = C[L]$ in let $x = C[M]$ in let $r = req(f, (x, \lambda_{y}y))$ in $r$ cf. (T-Req)</td>
</tr>
</tbody>
</table>
Compilation of $\lambda_{rpc}$-typing derivations into $\lambda_{rpc}^{enc}$

Direct style compilation for the client part, and CPS compilation for the server part

Server: $S[M_{rpc}] K = M_{rpc}^{enc}$

- $S[x] K = K(x)$
- $S[\lambda^c_x.M] K = K(\lambda^c_x.C[M])$
- $S[\lambda^s_x.M] K = K(\lambda^s(x, k).S[M] k)$
- $S[L^c M] K = S[L] (\lambda^s f. S[M] (\lambda^s x. call(\lambda^c_x.let y = f(x) in req(K, y), x )))$
- $S[L^s M] K = S[L] (\lambda^s f. S[M] (\lambda^s x. f(x, K))))$

$\Rightarrow$ Note call(-,-) is always in the tail position.
The semantics of $\lambda^{enc}_{rpc}$

Configuration (Conf): $Client \mid Server$

- Client: either a term $M$ or a client context $\Pi$
- Server: either a term $M$ or a server context $\Delta$

Client context $\Pi ::= \text{ctx } x \ M$ ($\approx \text{let } x = [ ] \text{ in } M$)

Server context $\Delta ::= \epsilon$

Evaluation step: $Conf \Rightarrow^{enc} Conf'$
The semantics of $\lambda_{rpc}^{enc}$

A session is either \((Req) \cdot (Call)\) or \((Req) \cdot (Reply)\), which corresponds to a single cycle of request-response on the client-server model.

Client:

(AppC) \(\text{let } y = (\lambda^c \bar{x}. M_0)(\bar{W}) \text{ in } M | \epsilon \Rightarrow^{enc} \text{let } y = M_0\{\bar{W}/\bar{x}\} \text{ in } M | \epsilon\)

(Req)\(^*\) \(\text{let } x = \text{req}(\lambda^s \bar{x}. M_0, \bar{W}) \text{ in } M | \epsilon \Rightarrow^{enc} \text{ctx } x \ M | (\lambda^s \bar{x}. M_0)(\bar{W})\)

(ValC) \(\text{let } x = V \text{ in } M | \epsilon \Rightarrow^{enc} M\{V/x\} | \epsilon\)

(LetC) \(\text{let } x = (\text{let } y = M_1 \text{ in } M_2) \text{ in } M | \epsilon \Rightarrow^{enc} \text{let } y = M_1 \text{ in } (\text{let } x = M_2 \text{ in } M) | \epsilon\)

Server:

(AppS) \(\Pi | (\lambda^s \bar{x}. M_0)(\bar{W}) \Rightarrow^{enc} \Pi | M_0\{\bar{W}/\bar{x}\}\)

(Call)\(^*\) \(\text{ctx } x \ M | \text{call}(\lambda^c \bar{x}. M_0, \bar{W}) \Rightarrow^{enc} \text{let } x = (\lambda^c \bar{x}. M_0)(\bar{W}) \text{ in } M | \epsilon\)

(Reply)\(^*\) \(\text{ctx } x \ M | V \Rightarrow^{enc} \text{let } x = V \text{ in } M | \epsilon\)
Slicing into the client and server parts

\[ \lambda_{rpc} \Rightarrow \lambda^{enc}_{rpc} \Rightarrow \lambda^{enc}_{cs} \]

The slicing compilation of \( \lambda^{enc}_{rpc} \) into \( \lambda^{enc}_{cs} \) \( \approx \) closure conversion

\[
CC[M] = m
\]

A state-encoding CS calculus \( \lambda^{enc}_{cs} \) with closure

Value \( v, w ::= x \mid \text{clo}(F, \bar{v}) \)

Term \( m ::= \cdots \)

Function store \( \phi_a ::= \{ \cdots , F = \bar{z}\lambda^a\bar{x}.m, \cdots \} \)

The semantics of \( \lambda^{enc}_{cs} \): \( Conf_1 \Rightarrow^{enc} Conf_2 \)

Correctness of the slicing compilation

- If \( M \Downarrow_c V \), then \( CC[C[M]] | \epsilon \Rightarrow^{enc*} CC[C[V]] | \epsilon \).
Compilation of the RPC term

Example: \((\lambda^s f. (\lambda^s x. x)^s_3 (f \ c_1 c)) \ s_1 (\lambda^c y. (\lambda^s z. z)^s_2 y)\)

\(\phi_c: \quad main = \text{let } r_3 = \text{req}_{s_1}(\text{clo}(g_7, \{}), \text{clo}(g_{10}, \{}), \text{clo}(g_{11}, \{})) \text{ in } r_3 \)
\(g_2 = \{f_7, f_5, k_4\} \lambda^c z_9. \text{let } r_{10} = f_7 \ z_9 \text{ in req}_{c_1}(\text{clo}(g_1, \{f_5, k_4\}, \ r_{10}) \)
\(g_{10} = \{\} \lambda^c y. \text{let } r_{14} = \text{req}_{s_2}(\text{clo}(g_8, \{}), \ y, \text{clo}(g_9, \{})) \text{ in } r_{14}\)

\(\phi_s: \quad g_1 = \{f_5, k_4\} \lambda^s x_6. f_5 (x_6, k_4) \)
\(g_3 = \{f_7, f_5, k_4\} \lambda^s x_8. \text{call}_{c_1}(\text{clo}(g_2, \{f_7, f_5, k_4\}), \ x_8) \)
\(g_4 = \{f_5, k_4\} \lambda^s f_7. \text{clo}(g_3, \{f_7, f_5, k_4\}) \ c \)
\(g_5 = \{k_4, f\} \lambda^s f_5. \text{clo}(g_4, \{f_5, k_4\})) \ f \)
\(g_6 = \{\} \lambda^s x, k_{11}. k_{11} \ x \)
\(g_7 = \{\} \lambda^s f, k_4. \text{clo}(g_5, \{k_4, f\}) \ s_3 (\text{clo}(g_6, \{})) \)
\(g_8 = \{\} \lambda^s z, k_{15}. k_{15} \ z \)
\(g_9 = \{\} \lambda^s x_{16}. x_{16} \)
\(g_{11} = \{\} \lambda^s x_{17}. x_{17} \)
A stateful implementation where some states may persist on the server during multiple subsequent cycles of request-response.

\[
\lambda_{rpc} \Rightarrow \lambda_{rpc}^{state} \Rightarrow \lambda_{cs}^{state}
\]
Motivation

In the following example, the cursor to a query result should persist on the server before and after the client function invocation:

For example,

\[ \lambda \text{query}. \text{let } \text{cursor} = \text{executeOnDatabase}(\text{query}) \text{ in } \]
\[ \text{let } \text{name} = \text{getNameFromRecord}(\text{cursor}) \text{ in } \]
\[ \text{let } r = f_{\text{client}}(\text{name}) \text{ in } \]
\[ \text{let } \text{cursor} = \text{nextRecord}(\text{cursor}) \text{ in } \cdots \]
Comparison with the state-encoding calculi

\[
\begin{align*}
\lambda^s f. \ (\lambda^s x. x) \ (f \ c) & \ (\lambda^c y. \ (\lambda^s z. z) \ y)
\end{align*}
\]
A stateful RPC calculus $\lambda_{rpc}^{state}$

$$\lambda_{rpc} \Rightarrow \lambda_{rpc}^{state} \Rightarrow \lambda_{cs}^{state}$$

In $\lambda_{rpc}^{state}$, remote procedure calls explicitly in term-level as:

Term $M :: \ V | \ let \ x = M \ in \ M | \ V_f(W) | \ req(V_f, W) | \ call(V_f, W) | \ ret(V)$

In the stateful semantics, a server stack $\Delta$ replaces $K$ as:

- "let $x = \text{ret}(V)$ in $M | \Delta$" in the stateful style instead of

  "let $x = \text{req}(K, V)$ in $M | \epsilon$" in the stateless style

In the compilation of $\lambda_{rpc}$ into $\lambda_{rpc}^{state}$,

- A typing derivation directed compilation

- Direct style compilation both for the client and the server
Compilation of $\lambda_{rpc}$-typing derivations into $\lambda_{rpc}^{state}$

Direct style compilation both for the client part and the server part

**Client:**  $C[M_{rpc}] = M_{rpc}^{state}$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C[x]$</td>
<td>$x$</td>
</tr>
<tr>
<td>$C[\lambda^c x. M]$</td>
<td>$\lambda^c x. C[M]$</td>
</tr>
<tr>
<td>$C[\lambda^s x. M]$</td>
<td>$\lambda^s x. S[M]$</td>
</tr>
</tbody>
</table>
| $C[L^c M]$ | $let f = C[L] in$
  $let x = C[M] in$
  $let r = f(x) in r$ |
| $C[L^s M]$ | $let f = C[L] in$
  $let x = C[M] in$
  $let r = req(f, x) in r$ | cf. (T-App) |
| | cf. (T-Req) |
Compilation of $\lambda_{rpc}$-typing derivations into $\lambda_{rpc}^{state}$

Direct style compilation both for the client part and the server part

Server: $S[M_{rpc}] = M_{rpc}^{state}$

$S[x] = x$

$S[\lambda^{c}x.M] = \lambda^{c}x.C[M]$  

$S[\lambda^{s}x.M] = \lambda^{s}x.S[M]$  

$S[L^{c}M] = \text{let } f = S[L] \text{ in } \begin{array}{l} \text{cf. (T-Call)} \\ \text{let } x = S[M] \text{ in } \\ \text{let } r = \text{call}(\lambda^{c}x.\text{let } y = f(x) \text{ in } \text{ret}(y), \ x) \text{ in } r \end{array}$

$S[L^{s}M] = \text{let } f = S[L] \text{ in } \begin{array}{l} \text{cf. (T-App)} \\ \text{let } x = S[M] \text{ in } \\ \text{let } r = f(x) \text{ in } r \end{array}$

$\Rightarrow$ Note call(-,-) can be both in the tail and non-tail positions.
The semantics of $\lambda_{\text{rpc}}^{\text{state}}$

Configuration (Conf): $\text{Client} \mid \text{Server}$
- Client: either a term $M$ or a client context $\Pi$
- Server: either a term $M$ or a server context stack $\Delta$

Client context $\Pi ::= \text{ctx} \times M$
Server context stack $\Delta ::= \epsilon \mid \text{ctx} \times M \cdot \Delta$

cf. $\text{ctx} \times M \approx \text{let } x = [\ ] \text{ in } M$

Evaluation step: $\text{Conf} \Rightarrow^{\text{state}} \text{Conf}'$
The semantics of $\lambda_{rpc}^{state}$

A session is $(\text{Req}) \cdot \{(\text{Call}) \cdot (\text{Ret})\}^{\text{zero or more}} \cdot (\text{Reply})$, and it may span multiple cycles of request-response on the client-server model.

Client:

(AppC) let $y = (\lambda^c \bar{x}.M_0)(\bar{W})$ in $M | \Delta$

$\Rightarrow^{state}$ let $y = M_0\{\bar{W}/\bar{x}\}$ in $M | \Delta$

(Req)* let $x = \text{req}(\lambda^s \bar{x}.M_0, \bar{W})$ in $M | \Delta$

$\Rightarrow^{state}$ ctx $x \ M | \Delta$; let $r = (\lambda^s \bar{x}.M_0)(\bar{W})$ in $r$

(ValC) let $x = V$ in $M | \Delta$ $\Rightarrow^{state} M\{V/x\} | \Delta$

(LetC) let $x = (\text{let } y = M_1 \text{ in } M_2)$ in $M | \Delta$

$\Rightarrow^{enc}$ let $y = M_1$ in (let $x = M_2$ in $M$) $| \Delta$

(Ret)* let $y = \text{ret}(V)$ in $M_2 | \text{ctx } x \ M_1 \cdot \Delta$ (cf. Pop)

$\Rightarrow^{state}$ ctx $y \ M_2 | \Delta$; let $x = V$ in $M_1$
The semantics of $\lambda_{rpc}^{state}$

Server:

(AppS) $\Pi \mid \Delta; \text{let } y = (\lambda^s x. M_0)(\overline{W}) \text{ in } M$

$\Rightarrow^{state} \Pi \mid \Delta; \text{let } y = M_0\{\overline{W}/\overline{x}\} \text{ in } M$

(Call)* $\text{ctx } y \ M_2 \mid \Delta; \text{let } x = \text{call}(\lambda^c x. M_0, \overline{W}) \text{ in } M_1$ (cf. Push)

$\Rightarrow^{state} \text{let } y = (\lambda^c x. M_0)(\overline{W}) \text{ in } M_2 \mid \text{ctx } x \ M_1 \cdot \Delta$

(Reply)* $\text{ctx } x \ M \mid \Delta; \ V \Rightarrow^{state} \text{let } x = V \text{ in } M \mid \Delta$

(ValS) $\Pi \mid \Delta; \text{let } x = V \text{ in } M \Rightarrow^{state} \Pi \mid \Delta; M\{V/x\}$

(LetS) $\Pi \mid \Delta; \text{let } x = (\text{let } y = M_1 \text{ in } M_2) \text{ in } M$

$\Rightarrow^{state} \Pi \mid \Delta; \text{let } y = M_1 \text{ in } (\text{let } x = M_2 \text{ in } M)$
Slicing into the client and server parts

\[ \lambda_{rpc} \Rightarrow \lambda^{state}_{rpc} \Rightarrow \lambda^{state}_{cs} \]

The slicing compilation of \( \lambda^{state}_{rpc} \) into \( \lambda^{state}_{cs} \) \((\approx \text{closure conversion})\)

\[ CC[M] = m \]

A stateful CS calculus \( \lambda^{state}_{cs} \)

Value

\[ v, w ::= x | clo(F, \bar{v}) \]

Term

\[ m ::= \cdots \]

Function store

\[ \phi_a ::= \{ \cdots, F = \bar{z}\lambda^{a}\bar{x}.m, \cdots \} \]

The semantics of \( \lambda^{state}_{cs}: \text{Conf}_1 \Rightarrow^{state} \text{Conf}_2 \)

Correctness of the slicing compilation

\[ \text{If } M \Downarrow_c V, \text{ then } CC[C[M]] | \epsilon \Rightarrow^{state*} CC[C[V]] | \epsilon. \]
Compilation of the RPC term

Example: \((\lambda^s f. (\lambda^s x. x) s_3 (f \ c_1 c)) s_1 (\lambda^c y. (\lambda^s z. z) s_2 y)\)

\(\phi_c : \quad \text{main} = \text{let } r_3 = \text{req}_{s_1}(\text{clo}(g_3, \{\}), \text{clo}(g_5, \{\})) \text{ in } r_3\)
\(g_2 = \{f_7\} \lambda^c z_{10}. \text{let } y_9 = f_7 \ z_{10} \text{ in ret}(y_9)\)
\(g_5 = \{} \lambda^c y. \text{let } r_{14} = \text{req}_{s_2}(\text{clo}(g_4, \{\}), y) \text{ in } r_{14}\)

\(\phi_s : \quad g_1 = \{} \lambda^s x. \ x\)
\(g_3 = \{} \lambda^s f. \text{let } x_5 = (\text{let } r_{11} = \text{call}_{c_1}(\text{clo}(g_2, \{f\}), c) \text{ in } r_{11}) \text{ in }\)
\(\text{let } r_6 = \text{clo}(g_1, \{\}) s_3 x_5 \text{ in } r_6\)
\(g_4 = \{} \lambda^s z. \ z\)
Discussion

An extended semantics for the session management

\[
\text{session} \ # \quad \text{nothing}
\]

- \(\text{client} | \text{server}\) or \(\text{client} | \text{server}\)

A mixed strategy

- employs the state-encoding calculi by default
- but switches to the use of the stateful calculi when necessary,
- separating the state-encoding part from the stateful part by the notion of monadic encapsulation of states

(Launchbury & Peyton Jones, 1994; Timany et al., 2017)
Related Work

Comparison with the original untyped RPC calculus

- Statically identified remote procedure calls
- A foundation for the stateful implementation in Links

<table>
<thead>
<tr>
<th></th>
<th>RPC calculi (single unified term)</th>
<th>CS(Client-Server) calculi (separate terms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>RPCs identified at type level</td>
<td>RPCs identified at term level</td>
</tr>
<tr>
<td></td>
<td>(\lambda_{rpc})</td>
<td>(\lambda^{enc}_{rpc})</td>
</tr>
<tr>
<td></td>
<td>(locative type system)</td>
<td>(stateless server)</td>
</tr>
<tr>
<td></td>
<td>typed compilation.</td>
<td>typed compilation.</td>
</tr>
<tr>
<td></td>
<td>(\lambda_{rpc})</td>
<td>(\lambda^{state}_{rpc})</td>
</tr>
<tr>
<td></td>
<td>(stateful server)</td>
<td>(stateful server)</td>
</tr>
<tr>
<td>Cooper&amp;Wadler's work</td>
<td>No attempt to identify RPCs</td>
<td>RPCs identified at term level</td>
</tr>
<tr>
<td></td>
<td>(\lambda_{rpc})</td>
<td>(\lambda_{cs})</td>
</tr>
<tr>
<td></td>
<td>(no type system)</td>
<td>(stateless server)</td>
</tr>
</tbody>
</table>
Related Work

In the (untyped) RPC calculus, the automatic slicing into the client and the server part into the CS (client-server) calculus:

- CPS conversion for stateless server
- Trampolined style for handling calling back from the server
  - An HTTP req-resp based asymmetric implementation
- Defunctionalisation for client and server closed procedures
Related Work

Programmer’s view and implementation model

- Symmetric communication vs. asymmetric Communication
- Client-server model vs. peer-to-peer model

<table>
<thead>
<tr>
<th>Symmetric communication</th>
<th>Client-server model</th>
<th>Peer-to-peer model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Links (Cooper et al, 2007)</td>
<td>Lambda5 (Murphy VII et al, 2004)</td>
<td></td>
</tr>
<tr>
<td>RPC (Cooper&amp;Wadler, 2009)</td>
<td>ML5 (Murphy, 2008)</td>
<td></td>
</tr>
<tr>
<td>Typed RPC</td>
<td>Multi-tier calculus</td>
<td></td>
</tr>
<tr>
<td>(Choi&amp;Chang, 2019)</td>
<td>(Neubauer &amp; Thiemann, 2005)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Asymmetric communication</th>
<th>Hop (Serrano &amp; Queinnec, 2010)</th>
<th>n/a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ur/Web (Chlipala, 2015)</td>
<td>Eliom (Radanne, 2017)</td>
<td></td>
</tr>
<tr>
<td>Eliom</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

A typed RPC calculus can statically discerns remote procedure calls providing type-based slicing with state-encoding and stateful calculi.

Future works

- Location polymorphic functions such as map
  \[ : \tau \rightarrow \tau' \text{ vs. } \tau \xrightarrow{a} \tau' \]

- A location inference method for locations at applications
  \[ : [M] \Gamma_1 \Rightarrow \tau_1 \sim \Gamma_2 \Rightarrow \tau_2 \text{ as a generalization of } [M]^{\tau_1 \sim \tau_2} \]

- Compilation of the two CS calculi into a session-typed calculus
  \[ : \text{Effects for communication } \phi ::= \tau \text{ chan } r | r!\tau | r?\tau | \cdots \]