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)

Introduction

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 λ-application for remote procedure calls

```
main^c = authenticate ()

authenticate = λ^s x.
 let creds = getCredentials "Enter name:passwd > " in
   if creds == "ezra:opensesame"
   then "the secret document" else "Access denied"

getCredentials = λ^c prompt. (print promt; 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

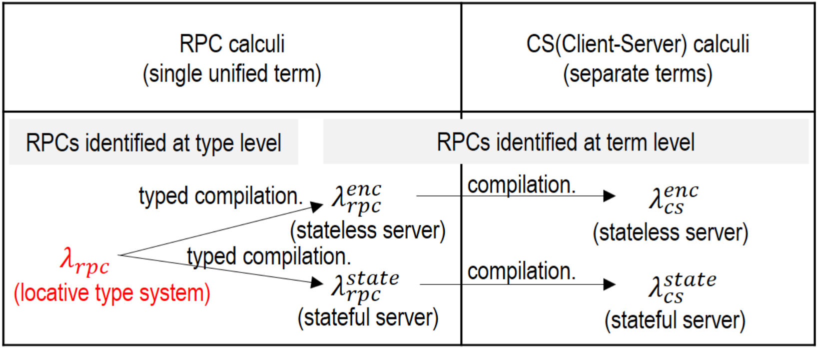
In this research

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



RPC calculi
(single unified term)

CS(Client-Server) calculi
(separate terms)

RPCs identified at type level

RPCs identified at term level

λ_{rpc}
(locative type system)

typed compilation.

λ_{rpc}^{enc}
(stateless server)

typed compilation.

λ_{rpc}^{state}
(stateful server)

compilation.

λ_{cs}^{enc}
(stateless server)

compilation.

λ_{cs}^{state}
(stateful server)

The RPC calculus

A call-by-value λ-calculus with location annotated λ-abstractions

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

$$\begin{array}{llcccc} \text{Location} & a, b & ::= & \mathbf{c} & \mid & \mathbf{s} \\ \\ \text{Term} & L, M, N & ::= & V & \mid & L\ M \\ \text{Value} & V, W & ::= & x & \mid & \lambda^a x.N \end{array}$$

Evaluation

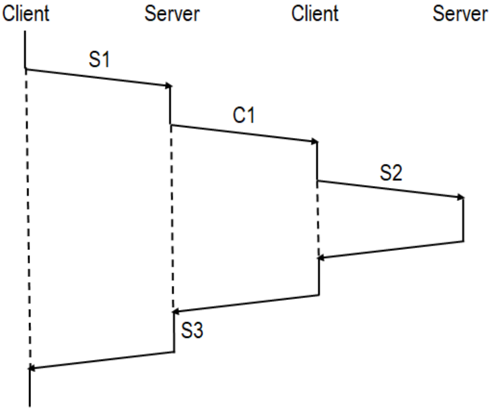
$$\frac{}{V \Downarrow_a V}\ \text{(Value)}$$

$$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{L\ M \Downarrow_a V}\ \text{(Beta)}$$

The RPC calculus

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

$$\overset{S1}{\boxed{(\lambda^s f.\ \overset{S3}{\boxed{(\lambda^s x.x)\ \overset{C1}{\boxed{(\, f\ c)}}}}\ )\ (\lambda^c y.\ \overset{S2}{\boxed{(\lambda^s z.z)\ \ y}}\ )}}$$



A locative type system for the RPC calculus

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

$$\text{Type} \quad \tau \quad ::= \quad \mathit{base} \quad | \quad \tau \xrightarrow{a} \tau$$

- $(\lambda^{\mathbf{s}} f.\ (\lambda^{\mathbf{s}} x.x)\ (f\ \ M))\ \ (\lambda^{\mathbf{c}} y.\ (\lambda^{\mathbf{s}} z.z)\ \ y)$

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

- $(\ \lambda^{\mathbf{c}} f.\ f\ \ M\ )\ \ (\text{if}\ \ \cdots\ \ \text{then}\ \lambda^{\mathbf{c}}\text{x.M}_1\ \text{else}\ \lambda^{\mathbf{s}}\text{y.M}_2)$

  Ill-typed because neither $f : \tau_1 \xrightarrow{\mathbf{c}} \tau_2$ nor $f : \tau_1 \xrightarrow{\mathbf{s}} \tau_2$

A locative type system for the RPC calculus

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

- A term M at location a has type τ under a type environment Γ

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

$$\frac{\Gamma \rhd_{a} L : \tau \xrightarrow{b} \tau' \qquad \Gamma \rhd_{a} M : \tau}{\Gamma \rhd_{a} L\ M : \tau'}$$

cf. $$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{L\ M \Downarrow_a V}\ \text{(Beta)}$$

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.

$$\text{(T-Var)}\ \frac{\Gamma(x) = \tau}{\Gamma \rhd_a x : \tau} \qquad \text{(T-Lam)}\ \frac{\Gamma\{x : \tau\} \rhd_b M : \tau'}{\Gamma \rhd_a \lambda^b x.M : \tau \xrightarrow{b} \tau'}$$

$$\text{(T-App)}\ \frac{\Gamma \rhd_a L : \tau \xrightarrow{a} \tau' \qquad \Gamma \rhd_a M : \tau}{\Gamma \rhd_a L\ M : \tau'}$$

$$\text{(T-Req)}\ \frac{\Gamma \rhd_{\mathbf{c}} L : \tau \xrightarrow{\mathbf{s}} \tau' \qquad \Gamma \rhd_{\mathbf{c}} M : \tau}{\Gamma \rhd_{\mathbf{c}} L\ M : \tau'}$$

$$\text{(T-Call)}\ \frac{\Gamma \rhd_{\mathbf{s}} L : \tau \xrightarrow{\mathbf{c}} \tau' \qquad \Gamma \rhd_{\mathbf{s}} M : \tau}{\Gamma \rhd_{\mathbf{s}} L\ M : \tau'}$$

A locative type system for the RPC calculus

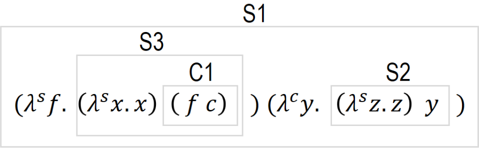
A example of a typing derivation where $f : base \xrightarrow{\mathbf{c}} base$,

$$(\lambda^{\mathbf{s}} f.\ (\lambda^{\mathbf{s}} x.x)\ {}^{\mathbf{s_3}}(f\ {}^{\mathbf{c_1}} c))\ {}^{\mathbf{s_1}}(\lambda^{\mathbf{c}} y.\ (\lambda^{\mathbf{s}} z.z)\ {}^{\mathbf{s_2}} y)$$

using the following notation:

- Notation: $L\ {}^{b}\ M$ for $\dfrac{\Gamma \rhd_{a} L : \tau \xrightarrow{b} \tau' \qquad \Gamma \rhd_{a} M : \tau}{\Gamma \rhd_{a} L\ M : \tau'}$

cf.



Properties of the locative type system

Type soundness for the RPC calculus

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

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

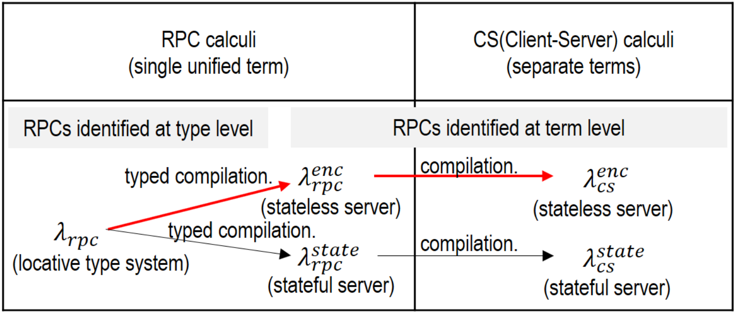
$[\![M]\!]^{\tau_1 \leadsto \tau_2}$: Location transformation of a term M of type τ_1 into τ_2

$$\begin{array}{lcl} [\![M]\!]^{\tau \leadsto \tau} & = & M \\ [\![M]\!]^{\tau_1 \xrightarrow{a} \tau_2 \ \leadsto \ \tau_3 \xrightarrow{b} \tau_4} & = & \lambda^b x.[\![M \ \ [\![x]\!]^{\tau_3 \leadsto \tau_1}]\!]^{\tau_2 \leadsto \tau_4} \end{array}$$

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

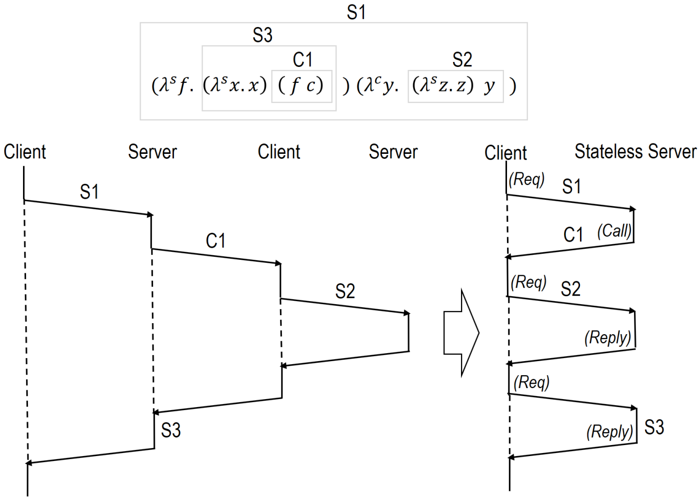
- $\lambda_{rpc} \quad \Rightarrow \quad \lambda_{rpc}^{enc} \quad \Rightarrow \quad \lambda_{cs}^{enc}$



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 λ^{enc}_{rpc}

$$\boxed{\lambda_{rpc} \quad \Rightarrow \quad \lambda^{enc}_{rpc} \quad \Rightarrow \quad \lambda^{enc}_{cs}}$$

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

$$\begin{array}{llll} \text{Term} & M & ::= & V \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \\ & & \mid & V_f(\overline{W}) \mid \mathsf{req}(V_f, \overline{W}) \mid \mathsf{call}(V_f, \overline{W}) \end{array}$$

In the compilation of λ_{rpc} into λ^{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 λ_{rpc}-typing derivations into λ_{rpc}^{enc}

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

$$\boxed{\text{Client:}\quad C[\![M_{rpc}]\!] \;=\; M_{rpc}^{enc}}$$

$$\begin{array}{lcll}
C[\![x]\!] & = & x & \\
C[\![\lambda^{\mathbf{c}}x.M]\!] & = & \lambda^{\mathbf{c}}x.C[\![M]\!] & \\
C[\![\lambda^{\mathbf{s}}x.M]\!] & = & \lambda^{\mathbf{s}}(x,k).S[\![M]\!]\ k & \\
C[\![L\ ^{\mathbf{c}}M]\!] & = & \text{let } f = C[\![L]\!] \text{ in} & \text{cf. (T-App)} \\
 & & \text{let } x = C[\![M]\!] \text{ in} & \\
 & & \text{let } r = f(x) \text{ in } r & \\
C[\![L\ ^{\mathbf{s}}M]\!] & = & \text{let } f = C[\![L]\!] \text{ in} & \text{cf. (T-Req)} \\
 & & \text{let } x = C[\![M]\!] \text{ in} & \\
 & & \text{let } r = \mathrm{req}(f,(x,\lambda^{\mathbf{s}}y.y)) \text{ in } r &
\end{array}$$

Compilation of λ_{rpc}-typing derivations into λ_{rpc}^{enc}

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

$$\boxed{\text{Server:}\quad S\llbracket M_{rpc}\rrbracket\ K \ = \ M_{rpc}^{enc}}$$

$$\begin{array}{lcll}
S\llbracket x\rrbracket\ K & = & K(x) & \\
S\llbracket \lambda^{\mathbf{c}}x.M\rrbracket\ K & = & K(\lambda^{\mathbf{c}}x.C\llbracket M\rrbracket) & \\
S\llbracket \lambda^{\mathbf{s}}x.M\rrbracket\ K & = & K(\lambda^{\mathbf{s}}(x,k).S\llbracket M\rrbracket\ k) & \\
S\llbracket L\ {}^{\mathbf{c}}M\rrbracket\ K & = & S\llbracket L\rrbracket\ (\lambda^{\mathbf{s}}f. & \text{cf. (T-Call)} \\
 & & \quad S\llbracket M\rrbracket\ (\lambda^{\mathbf{s}}x. & \\
 & & \qquad \text{call}(\ \lambda^{\mathbf{c}}x.\text{let } y = f(x) \text{ in } \text{req}(K,y), \quad x\ )\ )) & \\
S\llbracket L\ {}^{\mathbf{s}}M\rrbracket\ K & = & S\llbracket L\rrbracket\ (\lambda^{\mathbf{s}}f. & \text{cf. (T-App)} \\
 & & \quad S\llbracket M\rrbracket\ (\lambda^{\mathbf{s}}x. & \\
 & & \qquad f(x,K))) &
\end{array}$$

⇒ Note call(-,-) is always in the tail position.

The semantics of λ_{rpc}^{enc}

Configuration (Conf): $\mathit{Client} \mid \mathit{Server}$

- Client : either a term M or a client context Π
- Server : either a term M or a server context Δ

Client context Π ::= ctx x M ($\approx$ let $x = [\,]$ in M)

Server context Δ ::= ϵ

Evaluation step: $\mathit{Conf} \Rightarrow^{enc} \mathit{Conf}'$

The semantics of λ_{rpc}^{enc}

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

Client:

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

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

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

(LetC) $\quad \text{let } x = (\text{let } y = M_1 \text{ in } M_2) \text{ in } M \mid \epsilon$

$\qquad \Rightarrow^{enc} \text{let } y = M_1 \text{ in } (\text{let } x = M_2 \text{ in } M) \mid \epsilon$

Server:

(AppS) $\quad \Pi \mid (\lambda^{\mathbf{s}}\bar{x}.M_0)(\overline{W}) \quad \Rightarrow^{enc} \quad \Pi \mid M_0\{\overline{W}/\bar{x}\}$

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

(Reply)* $\quad \text{ctx } x\ M \mid V \quad \Rightarrow^{enc} \quad \text{let } x = V \text{ in } M \mid \epsilon$

Slicing into the client and server parts

$$\boxed{\lambda_{rpc} \quad \Rightarrow \quad \lambda_{rpc}^{enc} \quad \Rightarrow \quad \lambda_{cs}^{enc}}$$

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

$$CC[\![M]\!] \;=\; m$$

A state-encoding CS calculus λ_{cs}^{enc} with closure

Value	v, w	$::=$	$x \mid clo(F, \bar{v})$
Term	m	$::=$	$\cdots$
Function store	ϕ_a	$::=$	$\{\, \cdots, \; F = \bar{z}\lambda^a\bar{x}.m, \cdots \}$

The semantics of λ_{cs}^{enc}: $Conf_1 \quad \Rightarrow^{enc} \quad Conf_2$

Correctness of the slicing compilation

- If $M \Downarrow_{\mathbf{c}} V$, then $CC[\![C[\![M]\!]]\!] \mid \epsilon \Rightarrow^{enc*} CC[\![C[\![V]\!]]\!] \mid \epsilon$.

Compilation of the RPC term

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

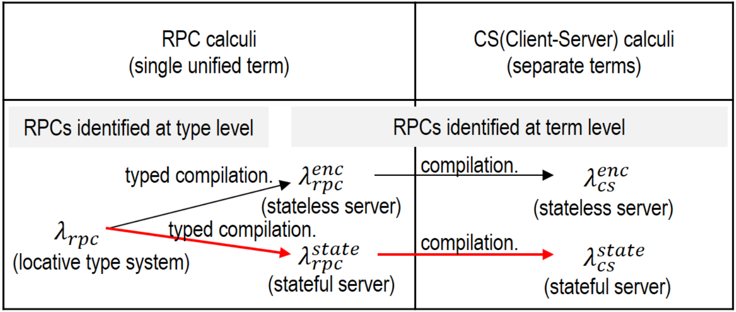
$$
\begin{array}{ll}
\phi_{\mathbf{c}}: & \mathit{main} \ = \ \mathrm{let}\ r_3 = \mathrm{req}_{s1}(\mathit{clo}(g_7, \{\}), \mathit{clo}(g_{10}, \{\}), \mathit{clo}(g_{11}, \{\}))\ \mathrm{in}\ r_3 \\
 & g_2 = \{f_7, f_5, k_4\}\ \lambda^{\mathbf{c}} z_9.\ \mathrm{let}\ r_{10} = f_7\ z_9\ \mathrm{in}\ \mathrm{req}_{c1}(\mathit{clo}(g_1, \{f_5, k_4\}),\ r_{10}) \\
 & g_{10} = \{\}\ \lambda^{\mathbf{c}} y.\ \mathrm{let}\ r_{14} = \mathrm{req}_{s2}(\mathit{clo}(g_8, \{\}),\ y, \mathit{clo}(g_9, \{\}))\ \mathrm{in}\ r_{14}
\end{array}
$$

$$
\begin{array}{ll}
\phi_{\mathbf{s}}: & g_1 = \{f_5, k_4\}\ \lambda^{\mathbf{s}} x_6.\ f_5\ (x_6, k_4) \\
 & g_3 = \{f_7, f_5, k_4\}\ \lambda^{\mathbf{s}} x_8.\ \mathrm{call}_{c1}(\mathit{clo}(g_2, \{f_7, f_5, k_4\}),\ x_8) \\
 & g_4 = \{f_5, k_4\}\ \lambda^{\mathbf{s}} f_7.\ \mathit{clo}(g_3, \{f_7, f_5, k_4\})\ c \\
 & g_5 = \{k_4, f\}\ \lambda^{\mathbf{s}} f_5.\ \mathit{clo}(g_4, \{f_5, k_4\}))\ f \\
 & g_6 = \{\}\ \lambda^{\mathbf{s}} x, k_{11}.\ k_{11}\ x \\
 & g_7 = \{\}\ \lambda^{\mathbf{s}} f, k_4.\ \mathit{clo}(g_5, \{k_4, f\})\ {}_{s3}\ (\mathit{clo}(g_6, \{\})) \\
 & g_8 = \{\}\ \lambda^{\mathbf{s}} z, k_{15}.\ k_{15}\ z \\
 & g_9 = \{\}\ \lambda^{\mathbf{s}} x_{16}.\ x_{16} \\
 & g_{11} = \{\}\ \lambda^{\mathbf{s}} x_{17}.\ x_{17}
\end{array}
$$

Part III: Slicing with stateful calculi

A stateful implementation where some states may persist on the server during multiple subsequents cycles of request-response

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



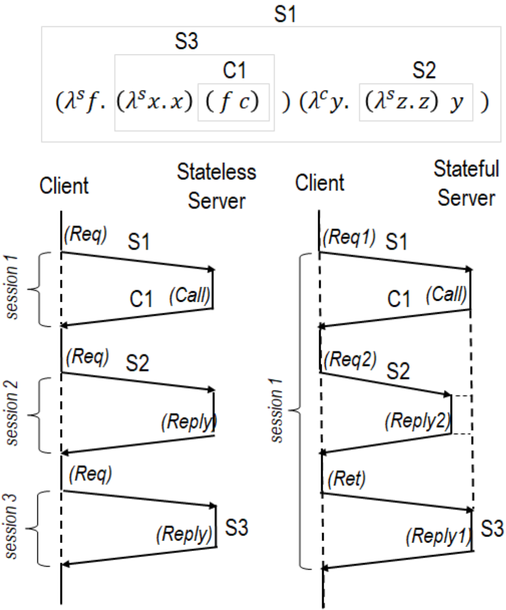
Motivation

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

For example,

$$
\begin{aligned}
\lambda^{\mathsf{s}} query.\ & \text{let } cursor = executeOnDatabase(query) \text{ in} \\
& \text{let } name = getNameFromRecord(cursor) \text{ in} \\
& \text{let } r = f_{client}(name) \text{ in} \\
& \text{let } cursor = nextRecord(cursor) \text{ in} \quad \cdots
\end{aligned}
$$

Comparison with the state-encoding calculi



A stateful RPC calculus λ_{rpc}^{state}

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

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

$$\begin{array}{llll} \text{Term} & M & ::= & V \mid \text{let } x = M \text{ in } M \\ & & \mid & V_f(\overline{W}) \mid \text{req}(V_f, \overline{W}) \mid \text{call}(V_f, \overline{W}) \mid \text{ret}(V) \end{array}$$

In the stateful semantics, a server stack Δ replaces K as:

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

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

In the compilation of λ_{rpc} into λ_{rpc}^{state},

- A typing derivation directed compilation
- Direct style compilation both for the client and the server

Compilation of λ_{rpc}-typing derivations into λ_{rpc}^{state}

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

$$\boxed{\text{Client:}\quad C[\![M_{rpc}]\!] \;=\; M_{rpc}^{state}}$$

$$\begin{array}{lcll}
C[\![x]\!] & = & x & \\
C[\![\lambda^{\mathbf{c}}x.M]\!] & = & \lambda^{\mathbf{c}}x.C[\![M]\!] & \\
C[\![\lambda^{\mathbf{s}}x.M]\!] & = & \lambda^{\mathbf{s}}x.S[\![M]\!] & \\
C[\![L\ ^{\mathbf{c}}M]\!] & = & \text{let } f = C[\![L]\!] \text{ in} & \text{cf. (T-App)} \\
 & & \text{let } x = C[\![M]\!] \text{ in} & \\
 & & \text{let } r = f(x) \text{ in } r & \\
C[\![L\ ^{\mathbf{s}}M]\!] & = & \text{let } f = C[\![L]\!] \text{ in} & \text{cf. (T-Req)} \\
 & & \text{let } x = C[\![M]\!] \text{ in} & \\
 & & \text{let } r = \text{req}(f, x) \text{ in } r &
\end{array}$$

Compilation of λ_{rpc}-typing derivations into λ_{rpc}^{state}

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

$$\boxed{\text{Server: } S\llbracket M_{rpc} \rrbracket \; = \; M_{rpc}^{state}}$$

$$\begin{array}{lcll}
S\llbracket x \rrbracket & = & x & \\
S\llbracket \lambda^{\mathbf{c}} x.M \rrbracket & = & \lambda^{\mathbf{c}} x.C\llbracket M \rrbracket & \\
S\llbracket \lambda^{\mathbf{s}} x.M \rrbracket & = & \lambda^{\mathbf{s}} x.S\llbracket M \rrbracket & \\
S\llbracket L \; ^{\mathbf{c}} M \rrbracket & = & \text{let } f = S\llbracket L \rrbracket \text{ in} & \text{cf. (T-Call)} \\
 & & \text{let } x = S\llbracket M \rrbracket \text{ in} & \\
 & & \text{let } r = \text{call}(\; \lambda^{\mathbf{c}} x.\text{let } y = f(x) \text{ in ret}(y), \; x \;) \text{ in } r & \\
S\llbracket L \; ^{\mathbf{s}} M \rrbracket & = & \text{let } f = S\llbracket L \rrbracket \text{ in} & \text{cf. (T-App)} \\
 & & \text{let } x = S\llbracket M \rrbracket \text{ in} & \\
 & & \text{let } r = f(x) \text{ in } r &
\end{array}$$

⇒ Note call(-,-) can be both in the tail and non-tail positions.

The semantics of λ_{rpc}^{state}

Configuration (Conf): *Client* | *Server*

- Client : either a term M or a client context Π
- Server : either a term M or a server context stack Δ

$$\begin{array}{lrcl} \text{Client context} & \Pi & ::= & \text{ctx}\ x\ M \\ \text{Server context stack} & \Delta & ::= & \epsilon \mid \text{ctx}\ x\ M \cdot \Delta \end{array}$$

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

Evaluation step: $Conf \Rightarrow^{state} Conf'$

The semantics of λ_{rpc}^{state}

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

Client:

(AppC) $\quad \text{let } y = (\lambda^{\mathbf{c}} \bar{x}.M_0)(\overline{W}) \text{ in } M \mid \Delta$
$\qquad \Rightarrow^{state} \quad \text{let } y = M_0\{\overline{W}/\bar{x}\} \text{ in } M \mid \Delta$

(Req)* $\quad \text{let } x = \text{req}(\lambda^{\mathbf{s}} \bar{x}.M_0, \overline{W}) \text{ in } M \mid \Delta$
$\qquad \Rightarrow^{state} \quad \text{ctx } x\ M \mid \Delta;\ \text{let } r = (\lambda^{\mathbf{s}} \bar{x}.M_0)(\overline{W}) \text{ in } r$

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

(LetC) $\quad \text{let } x = (\text{let } y = M_1 \text{ in } M_2) \text{ in } M \mid \Delta$
$\qquad \Rightarrow^{enc} \text{let } y = M_1 \text{ in } (\text{let } x = M_2 \text{ in } M) \mid \Delta$

(Ret)* $\quad \text{let } y = \text{ret}(V) \text{ in } M_2 \mid \text{ctx } x\ M_1 \cdot \Delta \qquad$ (cf. Pop)
$\qquad \Rightarrow^{state} \quad \text{ctx } y\ M_2 \mid \Delta;\ \text{let } x = V \text{ in } M_1$

The semantics of λ_{rpc}^{state}

Server:

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

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

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

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

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

Slicing into the client and server parts

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

The slicing compilation of λ_{rpc}^{state} into λ_{cs}^{state} ($\approx$ closure conversion)

$$CC[\![M]\!] \; = \; m$$

A stateful CS calculus λ_{cs}^{state}

Value	v, w	$::=$	$x \mid clo(F, \bar{v})$
Term	m	$::=$	$\cdots$
Function store	ϕ_a	$::=$	$\{ \cdots, \; F = \bar{z}\lambda^a\bar{x}.m, \cdots \}$

The semantics of λ_{cs}^{state}: $Conf_1 \quad \Rightarrow^{state} \quad Conf_2$

Correctness of the slicing compilation

- If $M \Downarrow_{\mathbf{c}} V$, then $CC[\![C[\![M]\!]]\!] \mid \epsilon \Rightarrow^{state*} CC[\![C[\![V]\!]]\!] \mid \epsilon$.

Compilation of the RPC term

Example: $(\lambda^{\mathbf{s}} f.\ (\lambda^{\mathbf{s}} x.x)\ ^{\mathbf{s}_3}(f\ ^{\mathbf{c}_1} c))\ ^{\mathbf{s}_1}(\lambda^{\mathbf{c}} y.\ (\lambda^{\mathbf{s}} z.z)\ ^{\mathbf{s}_2} y)$

$$\begin{array}{ll}
\phi_{\mathbf{c}}: & main \ = \ \mathrm{let}\ r_3 = \mathrm{req}_{s1}(clo(g_3,\{\}),\ clo(g_5,\{\}))\ \mathrm{in}\ r_3 \\
& g_2 = \{f_7\}\ \lambda^{\mathbf{c}} z_{10}.\ \mathrm{let}\ y_9 = f_7\ z_{10}\ \mathrm{in}\ \mathrm{ret}(y_9) \\
& g_5 = \{\}\ \lambda^{\mathbf{c}} y.\ \mathrm{let}\ r_{14} = \mathrm{req}_{s2}(clo(g_4,\{\}),\ y)\ \mathrm{in}\ r_{14}
\end{array}$$

$$\begin{array}{ll}
\phi_{\mathbf{s}}: & g_1 = \{\}\ \lambda^{\mathbf{s}} x.\ x \\
& g_3 = \{\}\ \lambda^{\mathbf{s}} f.\ \mathrm{let}\ x_5 = (\mathrm{let}\ r_{11} = \mathrm{call}_{c1}(clo(g_2,\{f\}),\ c)\ \mathrm{in}\ r_{11})\ \mathrm{in} \\
& \qquad\qquad\qquad \mathrm{let}\ r_6 = clo(g_1,\{\})\ {}_{s3}\ x_5\ \mathrm{in}\ r_6 \\
& g_4 = \{\}\ \lambda^{\mathbf{s}} z.\ z
\end{array}$$

Discussion

An extended semantics for the session management

- $client \overset{session\ \#}{|} server$ or $client \overset{nothing}{|} server$

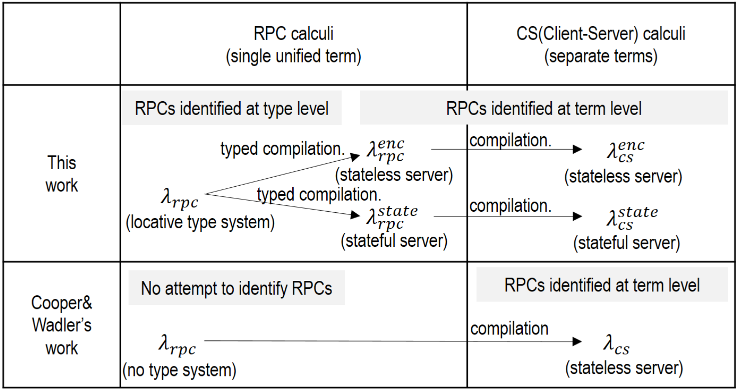
A mixed strategy

- 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 (Launchbuary & 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



	RPC calculi (single unified term)	CS(Client-Server) calculi (separate terms)
This work	RPCs identified at type level λ_{rpc} (locative type system) typed compilation. λ_{rpc}^{enc} (stateless server) typed compilation. λ_{rpc}^{state} (stateful server)	RPCs identified at term level compilation. λ_{cs}^{enc} (stateless server) compilation. λ_{cs}^{state} (stateful server)
Cooper& Wadler's work	No attempt to identify RPCs λ_{rpc} (no type system)	RPCs identified at term level compilation λ_{cs} (stateless server)

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

	Client-server model	Peer-to-peer model
Symmetric communication	Links (Cooper et al, 2007) RPC (Cooper&Wadler, 2009) Typed RPC (Choi&Chang, 2019)	Lambda5 (Murphy VII et al, 2004) ML5 (Murphy, 2008) Multi-tier calculus (Neubauer & Thiemann, 2005)
Asymmetric communication	Hop (Serrano & Queinnec, 2010) Ur/Web (Chlipala, 2015) Eliom (Radanne, 2017)	n/a

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
  : $\tau \to \tau'$ vs. $\tau \xrightarrow{a} \tau'$
- A location inference method for locations at applications
  : $[\![M]\!]^{\Gamma_1 \Rightarrow \tau_1 \rightsquigarrow \Gamma_2 \Rightarrow \tau_2}$ as a generalization of $[\![M]\!]^{\tau_1 \rightsquigarrow \tau_2}$
- Compilation of the two CS calculi into a session-typed calculus
  : Effects for communication $\varphi ::= \tau \text{ chan } r \mid r!\tau \mid r?\tau \mid \cdots$