A Polymorphic RPC Calculus

A Polymorphic RPC Calculus

Kwanghoon Choi[a,1], James Cheney[b,c,2], Simon Fowler[b], Sam Lindley[b,d,3]

[a] *Chonnam National University, Gwangju, Republic of Korea*
[b] *The University of Edinburgh, Scotland, UK*
[c] *The Alan Turing Institute, London, UK*
[d] *Imperial College, London, UK*

Abstract

The RPC calculus is a simple semantic foundation for multi-tier programming languages such as Links in which located functions can be written for the client-server model. Subsequently, the typed RPC calculus is designed to capture the location information of functions by types and to drive location type-directed slicing compilations. However, the use of locations is currently limited to monomorphic ones, which is one of the gaps to overcome to put into practice the theory of RPC calculi for client-server model.

This paper proposes a polymorphic RPC calculus to allow programmers to write succinct multi-tier programs using polymorphic location constructs. Then the polymorphic multi-tier programs can be automatically translated into programs only containing location constants amenable to the existing slicing compilation methods. We formulate a type system for the polymorphic RPC calculus, and prove its type soundness. Also, we design a monomorphization translation together with proofs on its type and semantic correctness for the translation.

Keywords: multi-tier programming, location polymorphism, remote procedure call, client-server model

1. Introduction

Multi-tier programming languages for the client-server model are designed to address the client-server dichotomy. For example, a web system basically consists of a web server that accesses databases and a web client that provides user interfaces, and they are connected by a network. Programmers have to develop two individual programs separately for the two machines, which increases the programmer's burden. Once a program is developed, programmers need to test

[1]This research was supported by the National Research Foundation of Korea grants from MSIP (No. 2017R1A2B4005138) and MoE (No. 2019R1I1A3A01058608).
[2]This work was supported by ERC Consolidator Grant Skye (grant number 682315).
[3]This work was supported by EPSRC Programme Grant "From Data Types to Session Types—A Basis for Concurrency and Distribution" (EP/K034413/1).

the two programs together, which is more complex than with one program on a single machine. After that, the integrity between the two programs should be properly maintained when each of them evolves. Further, some tasks cross the boundary of two computers, and so the separate development increases coupling between the client and server modules for the tasks.

Multi-tier programming attempts to solve this problem by allowing programmers to write a unified program for client and server expressions together in a single programming language, and by providing a slicing compilation method that can slice the unified program into separate client and server programs automatically.

The untyped RPC calculus (Cooper and Wadler, 2009) is the semantic foundation for the RPC (remote procedure call) feature of Links (Cooper et al., 2007), which is a functional programming language for multi-tier web programming. Here is an example excerpted from the RPC calculus paper, rewritten in Links as:

```
fun main() client { authenticate () }

fun authenticate() server {
   var creds = getCredentials( "Enter name:passwd > " )
   if ( creds == "ezra:opensesame" ) {
     "The secret document"
   } else { "Access denied" }
}

fun getCredentials(prompt) client { (print(prompt);  read) }
```

There are location attributes client and server that indicate where the associated functions should run. The program begins with *main*, which is a client function. It invokes a server function *authenticate*. Subsequently, in the body of the server function, a client function *getCredentials* is invoked. These are two examples of remote procedure call (RPC), one from the client to the server and the other from the server to the client. The RPC calculus thus expresses remote procedure calls as plain lambda applications. It also expresses local procedure calls such as *print*(*prompt*) as the same syntax of lambda application. As a result, it may take some time to see if a given lambda application is a remote procedure call, particularly when higher-order functions are extensively used. The two remote procedure calls in the example are underlined.

Then a slicing compilation in the untyped RPC calculus slices a unified program such as the one above into a client program and a server program where each separate program contains only functions that must run at their own location and the remote procedure calls are compiled with some communication primitives.

The typed RPC calculus (Choi and Chang, 2019) is an extension with location types to specify where functions must run. For example, `authenticate` has type $Unit \xrightarrow{\mathbf{s}} String$ while `getCredentials` has type $String \xrightarrow{\mathbf{c}} String$ where **s** denotes the server and **c** does the client. It is equipped with a type system that can account for remote procedure calls at the type level as is done in the previous example by underlining them. Also, it provides a type-directed slic-

ing compilation method simpler than the untyped slicing compilation method. Thanks to the simplicity, the method offers a spectrum of slicing compilations: one for the stateless server where no states are maintained in the server, which is good for scalability, and the other for the stateful server where the server maintains all states during multiple interactions with the client. The method even suggests an idea of how to mix the two styles. The details for these slicing compilations are in (Choi and Chang, 2019).

Note that in Links, location attributes are hints that are used at run-time rather than part of the type as in the typed RPC calculus.

In spite of the advancement, there are still gaps in putting into practice the theory of RPC calculi for client-server model. The typed RPC calculus is good for writing functions with specific locations such as web page modification and database accesses. But it is bad for writing location neutral functions such as list utilities and primitive type functions because programmers write them twice, one for the client and the other for the server. Instead, the calculus should allow programmers to write location neutral functions only once for the two locations. Then the compiler should be extended to translate them into location-specific versions, for example, one written in JavaScript for the client and the other written in OCaml for the server, automatically. It is much like the convenience of polymorphically typed functions that are written once but can be applied multiple times over different instantiated types.

An introduction of polymorphic locations to multi-tier programming languages including the RPC calculi poses a technical problem. In the RPC calculi, programmers write remote procedure calls in the same syntax as for local procedure calls. They provide no RPC keyword. This is believed to be a good design for programmers because this abstracts out a difference in terms of the use of client-server communication. For implementation, however, we need to distinguish between these two kinds of procedure calls, one implemented by a jump instruction and the other by a RPC library, exposing the difference explicitly in terms. The typed RPC calculus does this by location types as, for example, every application is a remote procedure call if the location of the application is different from the location of a function to invoke. On introducing polymorphic locations, we will have *location variables* where such location information is uncertain at compile-time.

In this paper, we propose a polymorphic RPC calculus, which is an extension of the typed RPC calculus with polymorphic locations. A key idea behind the polymorphic RPC calculus is to introduce location variables on lambda abstractions, to abstract locations by *location abstraction*, $\Lambda l.M$, and to instantiate them by *location application*, $M[Loc]$ for some location Loc. For example, the location-neutral `map` function could be written in Links extended with the feature of polymorphic locations as 'fun map(f, xs) l { the body of map }' where the location attribute is replaced with a location variable l. In other words,

$$map = \Lambda l.\lambda^l f.\lambda^l xs.\ \cdots the\ body\ of\ map \cdots$$

that has type $\forall l.(A \xrightarrow{l} B) \xrightarrow{l} ([A] \xrightarrow{l} [B])$. The type of f is $A \xrightarrow{l} B$, the type of xs

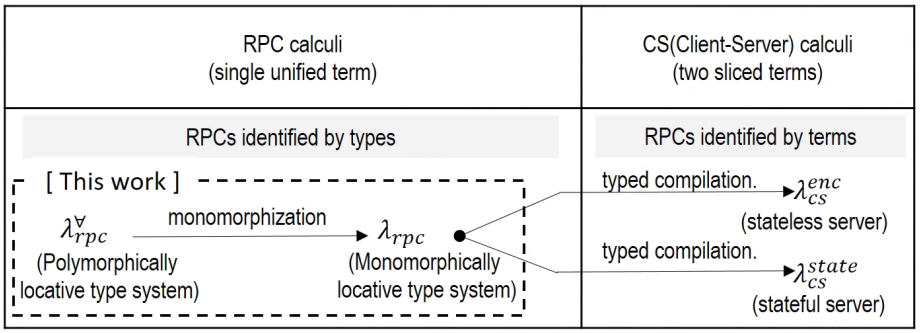

Figure 1: Overview of a polymorphic RPC calculus

is a list type whose elements have type A, i.e., $[A]$, and the ultimate return type is $[B]$. The map function should run at location l, which is specified by location application. To run it as a client function, we use a location application $map[\mathbf{c}]$, which becomes $\lambda^{\mathbf{c}} f.\lambda^{\mathbf{c}} xs.\ \cdots$ of type $(A \xrightarrow{\mathbf{c}} B) \xrightarrow{\mathbf{c}} ([A] \xrightarrow{\mathbf{c}} [B])$ by replacing all occurrences of l with $\mathbf{c}$. To run it as a server function, $map[\mathbf{s}]$ will be used. Thus, every polymorphic λ-abstraction can be regarded as a location-neutral one, and a choice of a location specific λ-abstraction is done by a location application with the location.

For implementation of the location polymorphism, we design a method to translate away both the location abstraction construct and the location application construct from polymorphic RPC terms. It then becomes possible to make use of the existing slicing compilation methods for the typed RPC calculus (Choi and Chang, 2019). Combining the translation and each of the existing slicing compilations, we can obtain two new slicing compilation methods for the polymorphic RPC calculus. Figure 1 shows an overview of the polymorphic RPC calculus.

The contributions of this paper are as follows:

- We propose a new polymorphic RPC calculus with the notion of polymorphic location, and prove its type soundness property.
- We design a monomorphization translation for the polymorphic RPC calculus, and prove the type correctness and the semantic correctness of the translation.

The roadmap of this paper is this. Section 2 reviews the typed RPC calculus as a background. In Section 3, we propose a polymorphic RPC calculus and a monomorphization translation, and prove a few important properties. After we discuss related work in Section 4, we conclude in Section 5.

2. Background: The typed RPC calculus

In this section, we review the typed RPC calculus. Figure 2 shows a typed RPC calculus λ_{rpc}. It is a call-by-value λ-calculus with location annotations

Syntax

Location	a, b	$::=$	$\mathbf{c}$	$\mid$	$\mathbf{s}$	
Term	L, M, N	$::=$	V	$\mid$	$L\ M$	$\mid$ $M[A]$
Value	V, W	$::=$	x	$\mid$	$\lambda^a x.M$	$\mid$ $\Lambda\alpha.V$

Semantics

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

$$\frac{}{\Lambda\alpha.V \Downarrow_a \Lambda\alpha.V}\ (\text{Tabs}) \qquad \frac{M \Downarrow_a \Lambda\alpha.V}{M[B] \Downarrow_a V\{B/\alpha\}}\ (\text{Tapp})$$

Figure 2: A typed RPC calculus λ_{rpc}

on λ-abstractions. The location annotations tell where the λ-abstractions must execute. The client-server model is assumed in the design of the calculus, and so the location annotations are either $\mathbf{c}$ denoting client or $\mathbf{s}$ denoting server. The syntax of the typed RPC calculus is thus defined as shown in the figure.

The semantics of λ_{rpc} is defined in a big-step operational semantics with evaluation judgment, $M \Downarrow_a V$ that denotes the evaluation of a term M at location a resulting in a value V. (Abs) straightforwardly defines an evaluation relation between a location annotated λ-abstraction and itself. (App) is more interesting for an application $L\ M$ at location a: it performs β-reduction at location b, which a λ-abstraction from L has as an annotation, with a value W from M, and it continues to evaluate the β-reduced term at the location. Here, $N\{W/x\}$ is a substitution of W for x in N.

Note that in (App), $L\ M$ is a remote procedure call whenever the caller location a is different from the callee location b. Otherwise, it is a local procedure call. When a is client and b is server, a server function is invoked from the client, and vice versa. The typed RPC calculus is a simple semantic foundation because it uses the same syntax of λ-application and the same evaluation rule (App) both for remote procedure calls and local ones. But every remote procedure call must be implemented differently from local procedure calls since it involves communication between client and server.

The typed RPC calculus in the figure is in fact an extension with polymorphic types of the original typed one (Choi and Chang, 2019), but is still limited to monomorphic locations in the same way. So, the typed RPC calculus now has type abstraction $\Lambda\alpha.V$ and type application $M[A]$ where α is a type variable and A is a type, which will be defined soon. (Tabs) and (Tapp) are quite the standard definitions for evaluation of type abstraction and type application. Note $V\{B/\alpha\}$ is a substitution of type B for type variable α in V.

A type system for the typed RPC calculus in Figure 3 basically comes from the one in (Choi and Chang, 2019) that accounts for remote procedure calls in the type level. It extends the original type system with polymorphic types by having two standard typing rules for type abstraction and type application, (T-Tabs) and (T-Tapp). $A\{B/\alpha\}$ is a substitution of B for each occurrence of α

Types

$$\text{Type} \quad A, B, C \quad ::= \quad \mathit{base} \quad | \quad A \xrightarrow{a} A \quad | \quad \alpha \quad | \quad \forall \alpha.A$$

Typing Rules

$$\text{(T-Var)}\ \frac{\Gamma(x) = A}{\Gamma \vdash_a x : A} \qquad \text{(T-Abs)}\ \frac{\Gamma, x : A \vdash_b M : B}{\Gamma \vdash_a \lambda^b x.M : A \xrightarrow{b} B}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash_a L : A \xrightarrow{b} B \qquad \Gamma \vdash_a M : A}{\Gamma \vdash_a L\ M : B}$$

$$\text{(T-Tabs)}\ \frac{\Gamma, \alpha \vdash_a V : A}{\Gamma \vdash_a \Lambda\alpha.V : \forall \alpha.A} \qquad \text{(T-Tapp)}\ \frac{\Gamma \vdash_a M : \forall \alpha.A}{\Gamma \vdash_a M[B] : A\{B/\alpha\}}$$

Figure 3: A type system for the typed RPC calculus

in A. Accordingly, every typing environment Γ now has type variables as well as associations of variables and types in general as $\{\alpha_1, \cdots, \alpha_k, x_1 : A_1, \cdots, x_m : A_m\}$.

The type system has two features related to location. First, location annotations are introduced to function types as $A \xrightarrow{a} B$. Every λ-abstraction that must run at location a gets this function type. For example, $(\lambda^{\mathbf{s}} f.\ (f\ M))\ \ (\lambda^{\mathbf{c}} y. \cdots)$ is well-typed when f is of type $A \xrightarrow{\mathbf{c}} B$ for some types A and B. However, $(\lambda^{\mathbf{c}} f.\ \cdots (\lambda^{\mathbf{c}} h.\ \cdots)\ f\ \ \cdots\ (\lambda^{\mathbf{c}} g.\ \cdots)\ f \cdots)$ is ill-typed when h is of $A \xrightarrow{\mathbf{c}} B$ and g is of $A \xrightarrow{\mathbf{s}} B$ since the type language of the typed RPC calculus only allows either $\mathbf{c}$ or $\mathbf{s}$, not both of them on a function type. In this respect, this typed RPC calculus (or the original one (Choi and Chang, 2019)) is monomorphic in terms of specifying location of evaluation.

Second, location annotations are also attached on typing judgments as $\Gamma \vdash_a M : A$ saying a term M at location a has type A under a type environment Γ. (T-Var) is defined as usual. (T-Abs) assigns λ-abstraction a function type with the same location as its annotation. Note that a location on the typing judgment in the conclusion changes to the annotated location in the premise for the body of λ-abstraction.

Combining these two features, (T-App) is designed to be a refinement of the conventional λ-application typing with respect to the combinations of location a (where to evaluate the application) and location b (where to evaluate the function). When a is different from b, $L\ M$ is statically found to be a remote procedure call: if $a = \mathbf{c}$ and $b = \mathbf{s}$, it is to invoke a server function from the client, and if $a = \mathbf{s}$ and $b = \mathbf{c}$, it is to invoke a client function from the server. Otherwise, one can statically decide that it is a local procedure call.

The type soundness theorem for the typed RPC calculus (Choi and Chang, 2019) guarantees that every remote procedure call thus identified statically will never change to a local procedure call under evaluation. The two slicing compilations for the typed RPC calculus depend on this capability of the type system as an analysis on dynamic communication patterns.

As shown in Figure 1, the typed RPC calculus offers slicing compilation

methods to slice a unified RPC program in λ_{rpc} into a client program and a server program in the client-server (CS) calculi λ_{cs}^{enc} or λ_{cs}^{state}, automatically. Each separate program will contain only functions that must run at one's own location, and the remote procedure calls in the unified program will be compiled with some communication primitives in the client-server programs.

The slicing methods are type-directed compilations where every input RPC program is type-checked to produce a typing derivation for it and then sliced programs are generated. In a unified RPC program, there is no particular syntax to specify remote procedure calls but only location types can be used to identify them. In separate client and server programs, we introduce explicit constructs for remote procedure calls like this. A construct $\mathsf{req}(V, W)$ is used to invoke a server function V with an argument W from the client, and $\mathsf{call}(V, W)$ is another remote procedure call construct in the reverse direction. In separate programs, we use $V(W)$ only for local procedure calls, differently from what we do in a unified program.

Now we are ready to present a key idea of the type-directed slicing compilations. Given a well-typed unified RPC program, each use of (T-App) in the typing derivation is compiled differently depending on the combination of the two locations on where to evaluate the application (a) and where to evaluate the function (b), as is explained above. When a is the same as b, the slicing compilation methods generate a normal application term, say, $V(W)$ where V is a local function and W is an argument. When $a = \mathbf{c}$ and $b = \mathbf{s}$, the slicing compilation methods generate $\mathsf{req}(V, W)$ where V is a server function. This term is implemented as sending V and W to the server to apply the function to the argument there and receiving either the application result or a new server-side call to invoke a client function. When $a = \mathbf{s}$ and $b = \mathbf{c}$, the slicing compilation methods generate $\mathsf{call}(V, W)$ where V is a client function. This term is implemented in a similar way for $\mathsf{req}(-, -)$ but in the reverse direction from the server to the client. In sliced programs, only applications of the form $V(W)$ do not involve communication at all. For details, the reader can refer to the formal semantics for the client-server calculi (λ_{cs}^{enc} and λ_{cs}^{state}) and the two slicing compilation rules in (Choi and Chang, 2019).

Although the original typed RPC calculus does not consider type polymorphism for a simple presentation, we see no problem in applying the two slicing compilations to this typed RPC calculus in the presence of type polymorphism assuming the target language also has type polymorphism.

3. A typed RPC calculus extended with polymorphic locations

In this section, we firstly extend the typed RPC calculus with the notion of polymorphic location to write polymorphic functions seamlessly with monomorphic functions, which is convenient for programmers. We call it a *polymorphic RPC calculus*, $\lambda_{rpc}^{\forall}$. Secondly, we design a translation of the polymorphic RPC calculus into the typed RPC calculus. Then we are able to make use of the two existing slicing compilation methods even for the polymorphic RPC calculus.

Syntax

Location	a, b	::=	$\mathbf{c}$	$\mid$	$\mathbf{s}$			
	Loc	::=	a	$\mid$	l			
Term	L, M, N	::=	V	$\mid$	$L\ M$	$\mid$ $M[A]$	$\mid$	$M[Loc]$
Value	V, W	::=	x	$\mid$	$\lambda^{Loc}x.M$	$\mid$ $\Lambda\alpha.V$	$\mid$	$\Lambda l.V$

Semantics

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

$$\frac{}{\Lambda\alpha.V \Downarrow_a \Lambda\alpha.V}\ (\text{Tabs}) \qquad \frac{M \Downarrow_a \Lambda\alpha.V}{M[B] \Downarrow_a V\{B/\alpha\}}\ (\text{Tapp})$$

$$\frac{}{\Lambda l.V \Downarrow_a \Lambda l.V}\ (\text{Labs}) \qquad \frac{M \Downarrow_a \Lambda l.V}{M[b] \Downarrow_a V\{b/l\}}\ (\text{Lapp})$$

Figure 4: The polymorphic RPC calculus $\lambda^{\forall}_{rpc}$

3.1. A polymorphic RPC calculus

An important feature of the polymorphic RPC calculus is the notion of location variable l for which we can substitute a location (constant) a. Accordingly, a new syntactic object Loc is introduced to be either a location constant or a location variable.

Every λ-abstraction $\lambda^{Loc}x.M$ now has an annotation of Loc instead of a. By substituting a location b for a location variable annotation, $(\lambda^l x.M)\{b/l\}$ becomes a monomorphic λ-abstraction $\lambda^b x.(M\{b/l\})$. This location variable is abstracted by the location abstraction construct $\Lambda l.V$, and it is instantiated by the location application construct $M[Loc]$. Except for these three constructs, the other syntax is the same as that in the typed RPC calculus. Figure 4 summarizes the syntax of the polymorphic RPC calculus.

The semantics for the polymorphic RPC calculus is shown in Figure 4. Every location abstraction is regarded as a value; it evaluates to itself by (Labs). Every location application $M[Loc]$ firstly evaluates to a location abstraction, and then Loc is substituted for the location variable in the body of the abstraction by (Lapp).

A new form of substitution $M\{Loc/l\}$ is defined as this.

$$\begin{aligned}
x\{Loc/l\} &= x \\
(\lambda^{Loc}x.M)\{Loc'/l\} &= \lambda^{Loc\{Loc'/l\}}x.M\{Loc'/l\} \\
(\Lambda\alpha.M)\{Loc/l\} &= \Lambda\alpha.(M\{Loc/l\}) \\
(\Lambda l.V)\{Loc/l'\} &= \begin{cases} \Lambda l.V & \text{if } l = l' \\ \Lambda l.(V\{Loc/l'\}) & \text{otherwise} \end{cases} \\
(L\ M)\{Loc/l\} &= (L\{Loc/l\})(M\{Loc/l\}) \\
(M[A])\{Loc/l\} &= (M\{Loc/l\})(A\{Loc/l\}) \\
(M[Loc])\{Loc'/l\} &= (M\{Loc'/l\})[Loc\{Loc'/l\}]
\end{aligned}$$

Types

$$\text{Type} \quad A, B, C \quad ::= \quad base \quad | \quad A \xrightarrow{Loc} B \quad | \quad \alpha \quad | \quad \forall \alpha.A \quad | \quad \forall l.A$$

Typing Rules

$$\text{(T-Var)}\ \frac{\Gamma(x) = A}{\Gamma \vdash_{Loc} x : A} \qquad \text{(T-Abs)}\ \frac{\Gamma, x : A \vdash_{Loc} M : B}{\Gamma \vdash_{Loc'} \lambda^{Loc} x.M : A \xrightarrow{Loc} B}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash_{Loc} L : A \xrightarrow{Loc'} B \quad \Gamma \vdash_{Loc} M : A}{\Gamma \vdash_{Loc} L\ M : B}$$

$$\text{(T-Tabs)}\ \frac{\Gamma, \alpha \vdash_{Loc} V : A}{\Gamma \vdash_{Loc} \Lambda\alpha.V : \forall\alpha.A} \qquad \text{(T-Tapp)}\ \frac{\Gamma \vdash_{Loc} M : \forall\alpha.A}{\Gamma \vdash_{Loc} M[B] : A\{B/\alpha\}}$$

$$\text{(T-Labs)}\ \frac{\Gamma, l \vdash_{Loc} V : A}{\Gamma \vdash_{Loc} \Lambda l.V : \forall l.A} \qquad \text{(T-Lapp)}\ \frac{\Gamma \vdash_{Loc} M : \forall l.A}{\Gamma \vdash_{Loc} M[Loc'] : A\{Loc'/l\}}$$

Figure 5: A type system for the polymorphic RPC calculus

where the definition of $Loc\{Loc'/l\}$ is

$$Loc\{Loc'/l\} \quad = \quad \begin{cases} Loc & \text{if } Loc = a \\ Loc' & \text{if } Loc = l' \text{ and } l = l' \\ Loc & \text{if } Loc = l' \text{ and } l \neq l' \end{cases}$$

and the definition of $A\{Loc/l\}$ will be presented below.

For example, a polymorphic identity function id is defined as $\Lambda l.\lambda^l x.x$ so that we can use $id[\mathbf{c}]$ as $\lambda^{\mathbf{c}} x.x$, and $id[\mathbf{s}]$ as $\lambda^{\mathbf{s}} x.x$. Thus, every polymorphic λ-abstraction can be regarded as a function usable both in the client and in the server, and a choice of a location specific λ-abstraction is done by a location application with the location.

Figure 5 shows a type system for the polymorphic RPC calculus. The type language allows function types to have the new syntactic object Loc as their annotation as $A \xrightarrow{Loc} B$. Then every λ-abstraction at unknown location gets assigned $A \xrightarrow{l} B$ using some location variable l. A universal quantifier over a location variable, $\forall l.A$, is also introduced to the type language accordingly.

We extend a typing judgment $\Gamma \vdash_{Loc} M : A$ with two things. First, a location variable can be annotated on it. This extension is used to assert a typing relation in the body of $\lambda^l x.M$. Second, typing environments now include location variables as well. A general form of Γ can now be written as $\{\alpha_1, \cdots, \alpha_k, l_1, \cdots, l_n, x_1 : A_1, \cdots, x_m : A_m\}$. This extension is used to keep track of a set of usable location variables. The domain of environment, $dom(\Gamma)$, is defined as a union of type, location, and term variables as $\{\alpha_1, \cdots, \alpha_k, l_1, \cdots, l_n, x_1, \cdots, x_m\}$, and the range, $rng(\Gamma)$, is $\{A_1, \cdots, A_m\}$.

Recall that $(\lambda^c f.\ \cdots (\lambda^c h.\ \cdots)\ f\ \ \cdots\ (\lambda^c g.\ \cdots)\ f \cdots)$ is an ill-typed term in the typed RPC calculus, where h is of type $A \xrightarrow{\mathbf{c}} B$ and g is of type $A \xrightarrow{\mathbf{s}} B$. The polymorphic RPC calculus can make the term be well-typed by assigning f

a polymorphic type as $\forall l.A \xrightarrow{l} B$ and by slightly changing the first and second occurrences of f into $f[\mathbf{c}]$ and $f[\mathbf{s}]$ respectively.

As a set of free variables is defined by the conventional definition of $fv(M)$, we define a set of free location variables over various kinds of objects in the form as $flv(-)$. Two definitions for locations and types are as follows.

$$\begin{aligned} flv(a) &= \{\} \\ flv(l) &= \{l\} \end{aligned}$$

$$\begin{aligned} flv(base) &= \emptyset \\ flv(\alpha) &= \emptyset \\ flv(A \xrightarrow{Loc} B)) &= flv(A) \cup flv(Loc) \cup flv(B) \\ flv(\forall \alpha.A) &= flv(A) \\ flv(\forall l.A) &= flv(A) \backslash \{l\} \end{aligned}$$

For typing environments, it is a union of location variables there and free location variables occurring in types associated with variables as

$$flv(\{\alpha_1, \cdots, \alpha_k, l_1, \cdots, l_n, x_1 : A_1, \cdots, x_m : A_m\}) = \{l_1, \cdots, l_n\} \cup \bigcup_{1 \leq i \leq m} flv(A_i)$$

The five typing rules (T-Var), (T-Abs), (T-App), (T-Tabs), and (T-Tapp) for the polymorphic RPC calculus generalize those for the typed RPC calculus by having Loc on function types and on typing judgments. Two new typing rules (T-Labs) and (T-Lapp) are similar to the typing rules for type abstraction and type application. (T-Labs) checks if the body of the location abstraction is typed with an extended typing environment with a fresh location variable. (T-Lapp) substitutes Loc' for all occurrences of a location variable l on λ-abstractions in M.

The definition of $A\{Loc/l\}$ is this.

$$\begin{aligned} base\{Loc/l\} &= base \\ (A \xrightarrow{Loc} B)\{Loc'/l\} &= A\{Loc'/l\} \xrightarrow{Loc\{Loc'/l\}} B\{Loc'/l\} \\ \alpha\{Loc/l\} &= \alpha \\ (\forall \alpha.A)\{Loc/l\} &= \forall \alpha.(A\{Loc/l\}) \\ (\forall l.A)\{Loc/l'\} &= \begin{cases} \forall l.A & \text{if } l = l' \\ \forall l.(A\{Loc/l'\}) & \text{otherwise} \end{cases} \end{aligned}$$

From now on, we will consider only well-formed typing judgments where there are no unbound variables, no unbound type variables, and no unbound free location variables. That is, given $\Gamma \vdash_{Loc} M : A$, we will safely assume three things. First, $fv(M) \subseteq dom(\Gamma)$. Second, $\bigcup_{A_i \in rng(\Gamma)} ftv(A_i) \cup ftv(M) \cup ftv(A) \subseteq dom(\Gamma)$ where $ftv(A)$ or $ftv(M)$ are the sets of free type variables

occurring in the type and the term respectively. They can be defined straightforwardly. Third, $\bigcup_{A_i \in rng(\Gamma)} flv(A_i) \cup flv(Loc) \cup flv(M) \cup flv(A) \subseteq dom(\Gamma)$. For example, in (T-App), location variables occurring in A and Loc' are from any location abstractions enclosing the application term, in (T-Tabs), a bound type variable α does not occur as a free type variable in Γ, and in (T-Labs), a bound location variable l never occurs as a free location variable in Γ and Loc.

3.1.1. Type soundness

Now we are ready to prove the type soundness of the type system for the polymorphic RPC calculus. Theorem 3.1 formulates this property. Its proof is done by induction on the height of evaluation derivations, and is available below. The value substitution lemma (3.2) offers a typing hypothesis for the β-reduced term necessary for proving the case (App) of the theorem. The type substitution lemma (3.3) proves a typing for a type substituted value from a type abstraction. The value relocation lemma (3.1) is used to prove that the movement of a return value from a callee location to a caller location does not change its type. The location substitution lemma (3.4) results in proving the case (Lapp) of the theorem by offering a typing hypothesis for the location-reduced term. The proofs of all these lemmas are available in the appendix.

Lemma 3.1 (Value relocation)**.** *If* $\Gamma \vdash_{Loc} V : A$ *then* $\Gamma \vdash_{Loc'} V : A$.

Lemma 3.2 (Value substitution)**.** *If* $\Gamma \vdash_{Loc} \lambda^{Loc'} x.M : A \xrightarrow{Loc'} B$ *and* $\Gamma \vdash_{Loc} V : A$ *then* $\Gamma \vdash_{Loc'} M\{V/x\} : B$.

Lemma 3.3 (Type substitution)**.** *If* $\Gamma \vdash_{Loc} \Lambda\alpha.V : \forall\alpha.A$ *then* $\Gamma \vdash_{Loc} V\{B/\alpha\} : A\{B/\alpha\}$.

Lemma 3.4 (Location substitution)**.** *If* $\Gamma \vdash_{Loc} \Lambda l.V : \forall l.A$ *then* $\Gamma \vdash_{Loc} V\{Loc'/l\} : A\{Loc'/l\}$.

Theorem 3.1 (Type soundness for $\lambda^{\forall}_{rpc}$)**.** *For a closed term* M*, if* $\emptyset \vdash_a M : A$ *and* $M \Downarrow_a V$*, then* $\emptyset \vdash_a V : A$.

Proof. We prove this theorem by induction on the height of the evaluation derivation. Base cases use (Abs), (Labs), and (Tabs) while inductive cases involve (App), (Lapp), and (Tapp).

Case (Abs): $\lambda^b x.M_0 \Downarrow_a \lambda^b x.M_0$ where $M = V = \lambda^b x.M_0$. Therefore, this case holds by the typing judgment hypothesis.

Case (Labs): $\Lambda l.V_0 \Downarrow_a \Lambda l.V_0$ where $M = V = \Lambda l.V_0$. Again, the typing judgment hypothesis proves this case.

Case (Tabs): $\Lambda\alpha.V_0 \Downarrow_a \Lambda\alpha.V_0$ where $M = V = \Lambda\alpha.V_0$. Once again, the typing judgment hypothesis proves this case.

Case (App): $L\ M_1 \Downarrow_a V$, (1):$L \Downarrow_a \lambda^b x.M_0$, (2):$M_1 \Downarrow_a W$, and (3):$M_0\{W/x\} \Downarrow_b V$ where $M = L\ M_1$. By $\emptyset \vdash_a L\ M_1 : B$, (4):$\emptyset \vdash_a L : A_1 \xrightarrow{Loc'} B$, and (5):$\emptyset \vdash_a M_1 : A_1$ where $B = A$.

By applying I.H. to (1) and (4), (6):$\emptyset \vdash_a \lambda^b x.M_0 : A_1 \xrightarrow{Loc'} B$ where $Loc' = b$. By applying I.H. to (2) and (5), (7):$\emptyset \vdash_a W : A_1$.

From (6) and (7), (8):$\emptyset \vdash_b M_0\{W/x\} : B$ is implied by the lemma (Value substitution).

By applying I.H to (3) and (8), (9):$\emptyset \vdash_b V : B$. By the lemma (Value relocation) with (9), $\emptyset \vdash_a V : B$. Since $B = A$, this case is proved.

Case (Lapp): $L[b] \Downarrow_a V_0\{b/l\}$, (1):$L \Downarrow_a \Lambda l.V_0$ where $M = L[b]$ and $V = V_0\{b/l\}$.

$\emptyset \vdash_a L[b] : A_0\{b/l\}$, (2):$\emptyset \vdash_a L : \forall l.A_0$ where $\Gamma = \emptyset$ and $A = A_0\{b/l\}$.

By applying I.H. to (1) and (2), (3):$\emptyset \vdash_a \Lambda l.V_0 : \forall l.A_0$.

The lemma (Location substitution) with (3) implies (4):$\emptyset \vdash_a V_0\{b/l\} : A_0\{b/l\}$, which proves this case.

Case (Tapp): $M_1[B] \Downarrow_a V_0\{B/\alpha\}$, (1):$M_1 \Downarrow_a \Lambda\alpha.V_0$ where $M = M_1[B]$ and $V = V_0\{B/\alpha\}$.

By $\emptyset \vdash_a M_1[B] : A_0\{B/\alpha\}$, (2):$\emptyset \vdash_a M_1 : \forall\alpha.A_0$ where $A = A_0\{B/\alpha\}$.

By applying I.H. to (1) and (2), (3):$\emptyset \vdash_a \Lambda\alpha.V_0 : \forall\alpha.A_0$.

By the lemma (Type substitution) with (3), $\emptyset \vdash_a V_0\{B/\alpha\} : A_0\{B/\alpha\}$, which proves this case.

□

3.1.2. An example using polymorphic location

To explain how polymorphic location is useful in writing multi-tier programs, we will present an example. Whenever a remote user enters text messages, the accumulated texts are converted to an HTML file for a local browser to display. For example, suppose that the user enters "Hi, Bobby!", "Your nose is shiny.", and "Where is John?", in sequence. Then the following HTML files will be generated in sequence:

- ⟨ul⟩⟨/ul⟩
- ⟨ul⟩⟨li⟩Hi, Bobby!⟨/li⟩⟨/ul⟩
- ⟨ul⟩⟨li⟩Hi, Bobby!⟨/li⟩ ⟨li⟩Your nose is shiny.⟨/li⟩⟨/ul⟩
- ⟨ul⟩⟨li⟩Hi, Bobby!⟨/li⟩ ⟨li⟩Your nose is shiny.⟨/li⟩ ⟨li⟩Where is John?⟨/li⟩⟨/ul⟩

Then the browser will display them one by one replacing each HTML file with the next one.

A program that behaves in this way may have type $Stream\ String \xrightarrow{\mathbf{c}} Stream\ HTML$ on a hypothetical runtime system, e.g.,

$$\lambda^{\mathbf{c}} prg.\ browser\ (prg\ keyboard)$$

where *keyboard* offers the program, *prg*, a stream of strings that the user enters, and *browser* displays a stream of HTMLs generated by the program.

Thus a simple minded multi-tier program is a conversion function of a string stream into an HTML stream. Such a program may be written by composing

a server function and a client function by a *cross-tier composition function* instantiated with the client location and the server location, $\circ[\mathbf{s},\mathbf{c}]$ (or written as $\circ_{[\mathbf{s},\mathbf{c}]}$ for readability), as this.

$$
\begin{array}{l}
program = (foldl_{[\mathbf{c}]}\ f\ [\,])\ \ (\circ_{[\mathbf{s},\mathbf{c}]})\ \ (foldl_{[\mathbf{s}]}\ g\ [\,[\,]\,]) \\
\quad \text{where} \\
\quad\quad f = \lambda^{\mathbf{c}} htmls.\lambda^{\mathbf{c}} strs.\ htmls\ +\!+_{[\mathbf{c}]} \\
\quad\quad\quad\quad\quad [\ \quad map_{[\mathbf{c}]}\ (\lambda^{\mathbf{c}} str.\ str)\ strs\quad \ ] \\
\quad\quad g = \lambda^{\mathbf{s}} strs.\lambda^{\mathbf{s}} str.\ strs\ +\!+_{[\mathbf{s}]}\ [\quad last_{[\mathbf{s}]}\ strs\ +\!+_{[\mathbf{s}]}\ [str]\quad]
\end{array}
$$

Note that the example uses [and] to denote a stream. It also assumes the standard stream functions extended with a single polymorphic location abstraction: the left fold function, $foldl$, the stream concatenation function, $(+\!+)$, the map function, *map*, and the last element selection function, *last*.

Also note that as the multi-tier programming languages such as Links, Ur/Web, and so on provide one's own convenient notation for HTML, the example makes use of such notation as $
\cdots
$ and $
\cdots
$ to construct HTML elements. They can be viewed as values of a special base type $HTML$.

In the structure of the program, the server function, $foldl_{[\mathbf{s}]}\ g\ [\,[\,]\,]$, is of type $Stream\ String \xrightarrow{\mathbf{s}} Stream\ Text$ where $Text$ is the type for a finite list of strings, and it generates a sequence of texts accumulated up to whenever a user enters a string. The client function, $foldl_{[\mathbf{c}]}\ f\ [\,]$, of type $Stream\ Text \xrightarrow{\mathbf{c}} Stream\ HTML$ takes the sequence of texts through the composition function. Each text, $[str_1, \cdots, str_n]$, is converted into an HTML, $
str_1, \cdots, str_n
$ for the browser to display, and the conversion is repeated over the sequence to update one HTML with the next one.

Now we are ready to review how the notion of polymorphic location is useful in writing multi-tier programs. First of all, polymorphic location functions are extensively used in the example. It is straightforward to use them at different locations just by instantiating them with the locations. The example requires no duplicate codes that would otherwise be written in the monomorphically typed RPC calculus, for example, for $foldl$, $(+\!+)$, *map*, and *last*. The resulting example looks more like a single computer program when all location applications are ignored, which is an ultimate goal in the multi-tier programming languages.

Second of all, it is also easy to compose two differently located functions as shown by $\circ[\mathbf{s},\mathbf{c}]$ in the example. The term and the type of the cross-tier composition function, $(\circ)$, can be defined as follows.

$$
\begin{array}{lcl}
\circ & : & \forall l_1, l_2.\forall \alpha, \beta, \gamma.(\beta \xrightarrow{l_2} \gamma) \xrightarrow{l_2} (\alpha \xrightarrow{l_1} \beta) \xrightarrow{l_2} (\alpha \xrightarrow{l_2} \gamma) \\
\circ & = & \Lambda l_1, l_2.\Lambda \alpha, \beta, \gamma.\lambda^{l_2} f.\lambda^{l_2} g.\lambda^{l_2} x.\ f\ (g\ x)
\end{array}
$$

Then $f \circ_{[\mathbf{s},\mathbf{c}]} g$ composes a server function g and a client function f with data flow from the server to the client. $f \circ_{[\mathbf{c},\mathbf{s}]} g$ does another cross-tier composition with the reverse data flow. Of course, it is also easy to express the local composition by $\circ[\mathbf{c},\mathbf{c}]$ and $\circ[\mathbf{s},\mathbf{s}]$. Without the notion of polymorphic location, four different located functions would be written, which is more laborious task.

Syntax

Location	a, b	$::=$	$\mathbf{c}$	$\mid$	$\mathbf{s}$			
Term	L, M, N	$::=$	V	$\mid$	$L\ M$	$\mid$	$M[A]$	
		$\mid$	(M, N)	$\mid$	$\pi_i(M)$			
Value	V, W	$::=$	x	$\mid$	$\lambda^a x.M$	$\mid$	$\Lambda\alpha.V$	$\mid$ (V, W)

Semantics

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

$$\frac{}{\Lambda\alpha.V \Downarrow_a \Lambda\alpha.V}\ (\text{Tabs}) \qquad \frac{M \Downarrow_a \Lambda\alpha.V}{M[B] \Downarrow_a V\{B/\alpha\}}\ (\text{Tapp})$$

$$\frac{L \Downarrow_a V \quad M \Downarrow_a W}{(L, M) \Downarrow_a (V, W)}\ (\text{Pair}) \qquad \frac{M \Downarrow_a (V_1, V_2) \quad i \in \{1, 2\}}{\pi_i(M) \Downarrow_a V_i}\ (\text{Proj-i})$$

Figure 6: The typed RPC calculus extended with pairs

3.2. A monomorphization translation of the polymorphic RPC calculus

Once programmers have the polymorphic RPC calculus in hand to be able to write RPC terms succinctly with the notion of polymorphic location, the next step is to design a method to compile to the client-server model. Instead of reinventing the wheel, we choose to make use of the two slicing compilation methods for the typed RPC calculus after translating polymorphic RPC terms into typed RPC terms. To realize this strategy, we need to translate all features of polymorphic locations away completely. This monomorphization translation will be discussed in this section.

A key idea behind the monomorphization translation is to interpret a location abstraction as a pair of its client version and its server version, and to regard a location application as a projection of the pair according to the location to be applied. For example, an identity function id, $\Lambda l.\lambda^l x.x$, is translated as $(\lambda^{\mathbf{c}} x.x, \lambda^{\mathbf{s}} x.x)$. Also, $id[\mathbf{c}]$ and $id[\mathbf{s}]$ are translated as $\pi_1([\![id]\!])$ and $\pi_2([\![id]\!])$, respectively, where $[\![M]\!]$ is a translation of M. A systematic translation like this allows the monomorphic typed RPC to pretend to have the notion of polymorphic locations without explicit location abstraction and location application.

Note that this pair-based translation scheme can potentially produce an exponentially large term. We believe that this would not be a big problem in practice as long as the nesting depth of location lambdas is shallow. This issue will be discussed more later.

For the target of the translation, we extend the typed RPC calculus with pairs (M, N) and projections $\pi_i(M)$ in the standard way as shown in Figure 6 and Figure 7. The extended type language includes pair types $A \times B$. From now on, we call this extension just the typed RPC calculus λ_{rpc}.

We present a monomorphization translation of $\lambda^{\forall}_{rpc}$ into λ_{rpc} as shown in Figure 8. The translation has two parts. The first part is for types. The type translation of *base* is *base*. We translate a monomorphic function type

Types

$$\text{Type} \quad A, B, C \quad ::= \quad \mathit{base} \quad | \quad A \xrightarrow{a} B \quad | \quad \alpha \quad | \quad \forall \alpha.A \quad | \quad A \times B$$

Typing Rules

$$\text{(T-Var)}\ \frac{\Gamma(x) = A}{\Gamma \vdash_a x : A} \qquad \text{(T-Abs)}\ \frac{\Gamma, x : A \vdash_b M : B}{\Gamma \vdash_a \lambda^b x.M : A \xrightarrow{b} B}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash_a L : A \xrightarrow{b} B \qquad \Gamma \vdash_a M : A}{\Gamma \vdash_a L\ M : B}$$

$$\text{(T-Tabs)}\ \frac{\Gamma, \alpha \vdash_a V : A}{\Gamma \vdash_a \Lambda\alpha.V : \forall\alpha.A} \qquad \text{(T-Tapp)}\ \frac{\Gamma \vdash_a M : \forall\alpha.A}{\Gamma \vdash_a M[B] : A\{B/\alpha\}}$$

$$\text{(T-Pair)}\ \frac{\Gamma \vdash_a L : A \qquad \Gamma \vdash_a M : B}{\Gamma \vdash_a (L, M) : A \times B}$$

$$\text{(T-Proj-i)}\ \frac{\Gamma \vdash_a M : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash_a \pi_i(M) : A_i}$$

Figure 7: A type system for the extended typed RPC calculus

with a location a element-wise leaving the location as it is. Any attempt to translate a function type with a location variable is undefined. We prevent this undesirable translation by systematically substituting $\mathbf{c}$ or $\mathbf{s}$ for every location variable that we meet during a translation and by starting from a closed type with no free location variables in it. The translation is designed to operate on closed types only with location constants all the time. The translation of polymorphic location types is $[\![\forall l.A]\!] = ([\![A\{\mathbf{c}/l\}]\!], [\![A\{\mathbf{s}/l\}]\!])$, which realizes the pair-based interpretation of location polymorphism as well as the invariant of location constants.

The second part is a translation of terms. The translations of variable, λ-abstraction, λ-application, type abstraction, and type application are done element-wise. The translation of location abstraction is defined as a pair-based interpretation with the first element for client and the second element for server: $[\![\Lambda l.V]\!] = ([\![V\{\mathbf{c}/l\}]\!], [\![V\{\mathbf{s}/l\}]\!])$. The translation maintains an invariant that whenever a term $\Lambda l.V$ is closed, the translated term also remains closed. This is so by systematically substituting $\mathbf{c}$ or $\mathbf{s}$ for every occurrence of the location variable l in the body V. The translation of location application with $\mathbf{c}$ or $\mathbf{s}$ is a projection of a pair that represents a polymorphic term. It is undefined when a location variable l appears as an argument in the location application. Our design associates the client version of the polymorphic term with the first element of the pair, and it does the server version with the second element. Therefore, $[\![M[a]]\!] = \pi_i([\![M]\!])$ where $i = 1$ if $a = \mathbf{c}$ and $i = 2$ if $a = \mathbf{s}$.

The monomorphization translation can be naturally extended for typing environments as: $[\![\{\alpha_1, \cdots, \alpha_k, x_1 : A_1, \cdots, x_n : A_n\}]\!] = \{\alpha_1, \cdots, \alpha_k, x_1 : [\![A_1]\!], \cdots, x_n : [\![A_n]\!]\}$. This translation is undefined if the typing environment contains any location variables or types associated with variables have any free

Translation: types

$[\![base]\!] = base$ $\quad [\![A \xrightarrow{a} B]\!] = [\![A]\!] \xrightarrow{a} [\![B]\!]$ $\quad [\![\forall l.A]\!] = [\![A\{\mathbf{c}/l\}]\!] \times [\![A\{\mathbf{s}/l\}]\!]$

$[\![\alpha]\!] = \alpha$ $\quad [\![\forall \alpha.A]\!] = \forall \alpha.[\![A]\!]$

Translation: terms

$[\![x]\!] = x$ $\quad [\![\lambda^a x.M]\!] = \lambda^a x.[\![M]\!]$ $\quad [\![\Lambda l.V]\!] = ([\![V\{\mathbf{c}/l\}]\!], [\![V\{\mathbf{s}/l\}]\!])$

$[\![L\ M]\!] = [\![L]\!]\ [\![M]\!]$ $\quad [\![M[\mathbf{c}]]\!] = \pi_1([\![M]\!])$ $\quad [\![M[\mathbf{s}]]\!] = \pi_2([\![M]\!])$

$[\![\Lambda \alpha.V]\!] = \Lambda \alpha.[\![V]\!]$ $\quad [\![M[B]]\!] = [\![M]\!][[\![B]\!]]$

Figure 8: A translation of the polymorphic RPC calculus into the RPC calculus

location variables.

3.2.1. Examples of the monomorphization translation

Let us discuss a few examples of the translation. The simplest one is with an identity function as:

$$\begin{aligned} [\![\Lambda l.\lambda^l x.x]\!] &= ([\![(\lambda^l x.x)\{\mathbf{c}/l\}]\!], [\![(\lambda^l x.x)\{\mathbf{s}/l\}]\!]) \\ &= (\lambda^{\mathbf{c}} x.[\![x]\!], \lambda^{\mathbf{s}} x.[\![x]\!]) \\ &= (\lambda^{\mathbf{c}} x.x, \lambda^{\mathbf{s}} x.x). \end{aligned}$$

Next are more complex examples about map functions over lists. Suppose M_X is a body of a map function parameterized by location arguments X as: *case xs of* { *nil* ⇒ *nil*; *cons y ys* ⇒ *cons* (*f y*) (*map*[X] *f ys*) } under some syntactic extension with case expression and with list constructors such as *nil* and *cons*. Also assume that $[A]$ is the list type over element type A.

Then a map function of type $\forall l.(A \xrightarrow{l} B) \xrightarrow{l} ([A] \xrightarrow{l} [B])$ can be defined by a recursive let construct as:

$$\texttt{letrec}\ \ map = \Lambda l.\lambda^l f.\lambda^l xs.M_l\ \ \texttt{in}\ \ \cdots$$

The translation of this map function proceeds as

$$\begin{aligned} [\![map]\!] &= ([\![(\lambda^l f.\lambda^l xs.M_l)\{\mathbf{c}/l\}]\!], [\![(\lambda^l f.\lambda^l xs.M_l)\{\mathbf{s}/l\}]\!]) \\ &= (\lambda^{\mathbf{c}} f.[\![\lambda^{\mathbf{c}} xs.M_{\mathbf{c}}]\!], \lambda^{\mathbf{s}} f.[\![\lambda^{\mathbf{s}} xs.M_{\mathbf{s}}]\!]) \\ &= (\lambda^{\mathbf{c}} f.\lambda^{\mathbf{c}} xs.[\![M_{\mathbf{c}}]\!], \lambda^{\mathbf{s}} f.\lambda^{\mathbf{s}} xs.[\![M_{\mathbf{s}}]\!]). \end{aligned}$$

To finish the translation of a recursive function such as *map*, we will ultimately need something like tying a knot over *map*[X] in the recursive call *map*[X] *f ys*. In the first element of the translated pair, *map*[$\mathbf{c}$] in $M_{\mathbf{c}}$ is translated as $\pi_1([\![map]\!])$, and, in the second element, *map*[$\mathbf{s}$] in $M_{\mathbf{s}}$ is translated as $\pi_2([\![map]\!])$. Here, instead of repeating $[\![map]\!]$ from the beginning, the translation stops here, and it just refers to the result of the beginning translation of *map* lazily to construct a recursive function. Then the translation continues over the rest of the body. As a result, we will get a pair of the client map function and the server map function.

Let us consider another map function of type $\forall l_1.\forall l_2.\forall l_3.(A \xrightarrow{l_3} B) \xrightarrow{l_1} ([A] \xrightarrow{l_2} [B])$ can be defined as:

$$\texttt{letrec}\ \ map = \Lambda l_1.\Lambda l_2.\Lambda l_3.map_0\ \ \texttt{in}\ \ \cdots$$

where map_0 is $\lambda^{l_1} f.\lambda^{l_2} xs.M_{l_1 l_2 l_3}$ and multiple location applications $M[Loc_1 \cdots Loc_k]$ is defined as $M[Loc_1] \cdots [Loc_k]$. The translation $[\![map]\!]$ becomes

$$\left(\begin{array}{c} \left(\begin{array}{c} ([\![map_0\{\mathbf{c}/l_1, \mathbf{c}/l_2, \mathbf{c}/l_3\}]\!], [\![map_0\{\mathbf{c}/l_1, \mathbf{c}/l_2, \mathbf{s}/l_3\}]\!]), \\ ([\![map_0\{\mathbf{c}/l_1, \mathbf{s}/l_2, \mathbf{c}/l_3\}]\!], [\![map_0\{\mathbf{c}/l_1, \mathbf{s}/l_2, \mathbf{s}/l_3\}]\!]) \end{array} \right), \\ \left(\begin{array}{c} ([\![map_0\{\mathbf{s}/l_1, \mathbf{c}/l_2, \mathbf{c}/l_3\}]\!], [\![map_0\{\mathbf{s}/l_1, \mathbf{c}/l_2, \mathbf{s}/l_3\}]\!]), \\ ([\![map_0\{\mathbf{s}/l_1, \mathbf{s}/l_2, \mathbf{c}/l_3\}]\!], [\![map_0\{\mathbf{s}/l_1, \mathbf{s}/l_2, \mathbf{s}/l_3\}]\!]) \end{array} \right) \end{array} \right)$$

which looks like a full binary tree with eight leaves. As you see, the monomorphization translation could generate exponentially many instances.

Let us discuss this problem in two ways. First, we believe that the first version of the map function would be typical in multi-tier programs by annotating the same location variable on all function types of a polymorphic type. For example, the Links compiler already performs something similar to the monomorphization for the standard library such as list utilities, which is common between the client and the server programs. It does code duplication for each common function as done with the first version, not for each argument of the functions as done with the second version. Our translation method can be regarded as a type-theoretic basis for the ad hoc code duplication in the Links compiler. In addition to this, the polymorphic RPC calculus offers a type-based method to control code duplication in a fine-grained way.

Note that the Eliom (Radanne, 2017; Radanne and Vouillon, 2018) compiler also uses code duplication to support a kind of location polymorphism, what they call *shared sections* in the expression-level and *mixed modules* in the module-level. In the related work, we will discuss these features in detail.

Second, we could take an alternative approach of dynamically passing locations even in separate client and server programs, which is radically different from the proposed static compilation of location polymorphism. This comes from the idea of compiling polymorphism using intensional type analysis on types arranged dynamically (Harper and Morrisett, 1995). It would solve the potential code explosion problem completely at the expense of runtime cost for maintaining location information and deciding whether to do local or remote calls dynamically. We will discuss this approach later.

3.2.2. Type correctness

Now we are ready to formulate the type correctness of the monomorphization translation as Theorem 3.2 saying that every closed well-typed term in $\lambda^{\forall}_{rpc}$ is translated to a well-typed term in λ_{rpc}. We prove this theorem by Lemma 3.7, which generalizes the formulation of the theorem to allow to have non-empty typing environments but only with free variables and free type variables, saying if $\Gamma \vdash_a M : A$ then $[\![\Gamma]\!] \vdash_a [\![M]\!] : [\![A]\!]$. This lemma should maintain an

invariant that the translation always sees type environments, terms, and types only with location constants. This invariant is sufficient to make sure that the three translations in the lemma are well-defined.

For this invariant, we need to define a set of free location variables for terms, $flv(M)$, as well. Here is a definition:

$$\begin{aligned} flv(x) &= \emptyset \\ flv(\lambda^{Loc}x.M) &= flv(Loc) \cup flv(M) \\ flv(\Lambda\alpha.V) &= flv(V) \\ flv(\Lambda l.V) &= flv(V)\backslash\{l\} \\ flv(L\ M) &= flv(L) \cup flv(M) \\ flv(M[A]) &= flv(M) \cup flv(A) \\ flv(M[Loc]) &= flv(M) \cup flv(Loc) \end{aligned}$$

The proof of Lemma 3.7 is done by induction on the height of a typing derivation for a term that the translation is to apply to. The invariant of location constants plays a role to ensure that the translation is well-defined for each subterm. The inductive proof works for all the cases except the case of (T-Tapp) and the case of (T-Labs). To finish the case of (T-Tapp), we use Lemma 3.5. This lemma turns a type substituted after a translation, which is obtained by the inductive hypothesis, into a translation of a substituted type, which is necessary to prove the case.

The case of (T-Labs) is more interesting. Dealing with $\Lambda l.V$, it needs two hypotheses over $V\{\mathbf{c}/l\}$ and $V\{\mathbf{s}/l\}$, not a hypothesis over V that is required by the plain inductive proof but violates the invariant because of the potential occurrence of a free location variable l. Lemma 3.6 allows the proof to use the two hypotheses as follows.

According to Lemma 3.6, given a typing derivation possibly with the use of free location variables, one can build a new typing derivation with the similar structure and the same height by substituting any of $\mathbf{c}$ or $\mathbf{s}$ for the occurrences of the free location variables in the typing derivation. This lemma accounts for location polymorphism in the polymorphic RPC calculus, which is why we call it the location polymorphism lemma.

The proof of Lemma 3.5 and Lemma 3.6 is available in the appendix.

Lemma 3.5 (Type substitution over type under monomorphization)**.** $[\![A]\!]\{[\![B]\!]/\alpha\} = [\![A\{B/\alpha\}]\!]$.

Lemma 3.6 (Location polymorphism)**.** *Suppose* $\Gamma = \{l_1, \cdots, l_n\} \cup \Gamma_0$ *such that* Γ_0 *has no location variables. If* $\Gamma \vdash_{Loc} M : A$ *then* $(\Gamma_0 \vdash_{Loc} M : A)\{a_1/l_1, \cdots, a_n/l_n\}$ *for any* $a_1, \cdots, a_n$ *with the same height.*

Lemma 3.7. *Given* $flv(\Gamma) \cup flv(M) \cup flv(A) = \emptyset$*, if* $\Gamma \vdash_a M : A$ *then* $[\![\Gamma]\!] \vdash_a [\![M]\!] : [\![A]\!]$.

Proof. We prove this lemma by induction on the height of the typing derivation. A base case uses (T-Var), and inductive cases involve the other kinds of typing rules.

(T-Var): $[\![\Gamma]\!]$ and $[\![A]\!]$ are well-defined since $FLV(\Gamma) = \emptyset$ and $FLV(A) = \emptyset$. By the typing derivation, $\Gamma(x) = A$. This implies (1):$[\![\Gamma]\!](x) = [\![A]\!]$. By (T-Var) with (1), $[\![\Gamma]\!] \vdash_a x : [\![A]\!]$, which proves this case since $[\![x]\!] = x$.

(T-Abs): $\Gamma \vdash_a \lambda^{Loc} x.M_0 : A_0 \xrightarrow{Loc} B_0$, (1):$\Gamma, x : A_0 \vdash_{Loc} M_0 : B_0$ where $M = \lambda^{Loc} x.M_0$ and $A = A_0 \xrightarrow{Loc} B_0$. It is easy to verify (2):$flv(\Gamma, x : A_0) \cup flv(M_0) \cup flv(B_0) = \emptyset$ by the hypothesis on free location variables.

I.H. can be applied to to (1) and (2) since $Loc = b$ for some b; $flv(A_0 \xrightarrow{Loc} B_0) = \emptyset$ implies $flv(Loc) = \emptyset$. This implies (3):$[\![\Gamma, x : A_0]\!] \vdash_{Loc} [\![M_0]\!] : [\![B_0]\!]$.

By (T-Abs) with (3), (4):$[\![\Gamma]\!] \vdash_a \lambda^{Loc} x.[\![M_0]\!] : [\![A_0]\!] \xrightarrow{Loc} [\![B_0]\!]$. By the definition of the translation, (4) implies $[\![\Gamma]\!] \vdash_a [\![\lambda^{Loc} x.M_0]\!] : [\![A_0 \xrightarrow{Loc} B_0]\!]$ since $Loc = b$.

(T-App): $\Gamma \vdash_a L\ M_0 : A$, (1):$\Gamma \vdash_a L : B \xrightarrow{Loc'} A$, and (2):$\Gamma \vdash_a M_0 : B$ where $M = L\ M_0$.

By well-formed typing judgments, $flv(B) = \emptyset$ and $flv(Loc') = \emptyset$.

By the argument stated above, we can safely assume (3):$flv(\Gamma) \cup flv(L) \cup flv(B \xrightarrow{Loc'} A) = \emptyset$. Also, we have (4):$flv(\Gamma) \cup flv(M_0) \cup flv(B) = \emptyset$.

By applying I.H. to (1) and (3), (5):$[\![\Gamma]\!] \vdash_a [\![L]\!] : [\![B \xrightarrow{Loc'} A]\!]$.

By applying I.H. to (2) and (4), (6):$[\![\Gamma]\!] \vdash_a [\![M_0]\!] : [\![B]\!]$.

Since $flv(Loc') = \emptyset$, $Loc' = b$ for some b. Therefore, $[\![B \xrightarrow{Loc'} A]\!] = [\![B]\!] \xrightarrow{Loc'} [\![A]\!]$. By (T-App) with (5) and (6), $[\![\Gamma]\!] \vdash_a [\![L]\!][\![M_0]\!] : [\![A]\!]$, which proves this case by $[\![M]\!] = [\![LM_0]\!] = [\![L]\!][\![M_0]\!]$.

(T-Tabs): $\Gamma \vdash_a \Lambda\alpha.V_0 : \forall\alpha.A_0$, and (1):$\Gamma, \alpha \vdash_a V_0 : A_0$ where $M = \Lambda\alpha.V_0$ and $A = \forall\alpha.A_0$.

(2):$flv(\Gamma, \alpha) \cup flv(V_0) \cup flv(A_0) = \emptyset$.

By applying I.H. to (2), $[\![\Gamma, \alpha]\!] \vdash_a [\![V_0]\!] : [\![A_0]\!]$, which is (3):$[\![\Gamma]\!], \alpha \vdash_a [\![V_0]\!] : [\![A_0]\!]$ by the def. of the translation.

By (T-Tabs) with (3), $[\![\Gamma]\!] \vdash_a \Lambda\alpha.[\![V_0]\!] : \forall\alpha.[\![A_0]\!]$. This is the same as $[\![\Gamma]\!] \vdash_a [\![\Lambda\alpha.V_0]\!] : [\![\forall\alpha.A_0]\!]$ by the def. of the translation, which proves this case.

(T-Tapp): $\Gamma \vdash_a M_1[B] : A_0\{B/\alpha\}$, and (1):$\Gamma \vdash_a M_1 : \forall\alpha.A_0$ where $M = M_1[B]$ and $A = A_0\{B/\alpha\}$.

(2):$flv(\Gamma) \cup flv(M_1) \cup flv(\forall\alpha.A_0) = \emptyset$.

By I.H. with (2), $[\![\Gamma]\!] \vdash_a [\![M_1]\!] : [\![\forall\alpha.A_0]\!]$, which is (3):$[\![\Gamma]\!] \vdash_a [\![M_1]\!] : \forall\alpha.[\![A_0]\!]$ by the def. of the translation.

By (T-Tapp) with(3),(4):$[\![\Gamma]\!] \vdash_a [\![M_1]\!][[\![B]\!]] : [\![A_0]\!]\{[\![B]\!]/\alpha\}$.

By the lemma 3.5 with (4), (5):$[\![\Gamma]\!] \vdash_a [\![M_1]\!][[\![B]\!]] : [\![A_0\{B/\alpha\}]\!]$, which proves this case by the def. of the translation.

(T-Labs): $\Gamma \vdash_a \Lambda l.V_0 : \forall l.A_0$, and (1):$\Gamma, l \vdash_a V_0 : A_0$ where $M = \Lambda l.V_0$ and $A = \forall l.A_0$.

By applying the lemma 3.6 to (1), (2): $\Gamma \vdash_a V_0\{a/l\} : A_0\{a/l\}$ for all a. Also, (3):$flv(\Gamma) \cup flv(V_0\{a/l\}) \cup flv(A_0\{a/l\}) = \emptyset$ since l is the only free location variable.

By applying I.H. to (2) and (3), (4):$[\![\Gamma]\!] \vdash_a [\![V_0\{a/l\}]\!] : [\![A_0\{a/l\}]\!]$ for $a = \mathbf{c}, \mathbf{s}$.

By (T-Pair) in the type system for λ_{rpc} with two premises, one with (4) and $a = \mathbf{c}$ and the other with (4) and $a = \mathbf{s}$, $[\![\Gamma]\!] \vdash_a ([\![V_0\{\mathbf{c}/l\}]\!], [\![V_0\{\mathbf{s}/l\}]\!]) : [\![A_0\{\mathbf{c}/l\}]\!] \times [\![A_0\{\mathbf{s}/l\}]\!]$. By the definition of the translation, $[\![\Gamma]\!] \vdash_a [\![\Lambda l.V_0]\!] : [\![\forall l.A_0]\!]$, which proves this case.

(T-Lapp): $\Gamma \vdash_a L[Loc'] : A_0\{Loc'/l\}$, and (1):$\Gamma \vdash_a L : \forall l.A_0$ where $M = L[Loc']$, $A = A_0\{Loc'/l\}$. $Loc' = b$ for some b since $flv(Loc') = \emptyset$ by the hypothesis on free location variables. Also, (2):$flv(\Gamma) \cup flv(L) \cup flv(\forall l.A_0) = \emptyset$ since $flv(A_0\{Loc'/l\}) = \emptyset$ and so l can be the only free location variable in A_0.

By applying I.H. to (1) and (2), $[\![\Gamma]\!] \vdash_a [\![L]\!] : [\![\forall l.A_0]\!]$, which is (3): $[\![\Gamma]\!] \vdash_a [\![L]\!] : [\![A_0\{\mathbf{c}/l\}]\!] \times [\![A_0\{\mathbf{s}/l\}]\!]$ by the definition of the translation.

By (T-Proj-i) with (3), $[\![\Gamma]\!] \vdash_a \pi_i([\![L]\!]) : [\![A_0\{Loc'/l\}]\!]$ where $i = 1$ if $Loc' = \mathbf{c}$, and $i = 2$ if $Loc' = \mathbf{s}$, which is (4):$[\![\Gamma]\!] \vdash_a [\![L[Loc']]\!] : [\![A_0\{Loc'/l\}]\!]$ by the definition of the translation. (4) proves this case. □

Theorem 3.2 (Type correctness of the monomorphization translation)**.** *For a closed term M, if $\emptyset \vdash_a M : A$ then $\emptyset \vdash_a [\![M]\!] : [\![A]\!]$.*

Proof. This theorem is proved by the lemma 3.7 as M is a top-level closed term and so $flv(\emptyset) \cup flv(M) \cup flv(A) = \emptyset$. □

3.2.3. Semantic correctness

This time we formulate a stronger property, the semantic correctness of the monomorphization translation, as Theorem 3.3: for a well-typed closed term, the evaluation of the term in $\lambda^{\forall}_{rpc}$ is preserved in λ_{rpc} under the translation. We prove this theorem by induction on the height of the evaluation derivation for the closed well-typed term. Two of the three base cases are (Abs) and (Tabs), which are proved immediately. The case (App) of the proof uses Lemma 3.8 saying the translation of a substitution is the same as a substitution of the translated terms. Similarly, the case (Tapp) uses Lemma 3.9 to turn a translated term substituted with a translated type into a translation of a term substituted with a type. The most interesting cases are (Labs) and (Lapp).

(Labs) is the remaining one of the three base cases, and it needs to prove that if $\Lambda l.V \Downarrow_a \Lambda l.V$, then $[\![\Lambda l.V]\!] \Downarrow_a [\![\Lambda l.V]\!]$. The proof in this case needs the second induction because $[\![\Lambda l.V]\!]$ becomes translated into a possibly deeply nested pair and the evaluation derivation of the translated term grows accordingly. Generally, $\Lambda l.V$ as a closed value is in the form of $\Lambda l.\Lambda l_1.\cdots\Lambda l_n.V_n$ where $V_n = \lambda^{Loc}x.M_0$ or $V_n = \Lambda\alpha.W$ due to the syntactic restriction on V. When n is zero, the translated term is a flat pair. When n is greater than zero, the translated pair looks more or less like a full binary tree of height n. By the second induction, the proof constructs an evaluation derivation of the same as the structure of the translated term of this case.

In (Lapp), we prove that if $L[b] \Downarrow_a V_0\{b/l\}$ then $[\![L[b]]\!] \Downarrow_a [\![V_0\{b/l\}]\!]$ under the assumption that $L \Downarrow_a \Lambda l.V_0$. The proof involves two subcases, one with $b = \mathbf{c}$ and the other with $b = \mathbf{s}$, both of which can be proved with no difficulty.

The proof of Lemma 3.8 and Lemma 3.9 is available in the appendix.

Lemma 3.8 (Substitution under monomorphization)**.** $[\![M]\!]\{[\![W]\!]/x\} = [\![M\{W/x\}]\!]$.

Lemma 3.9 (Type substitution over term under monomorphization)**.** $[\![V]\!]\{[\![B]\!]/\alpha\} = [\![V\{B/\alpha\}]\!]$.

Theorem 3.3 (Semantic correctness of monomorphization)**.** *If* $\emptyset \vdash_a M : A$ *and* $M \Downarrow_a V$*, then* $[\![M]\!] \Downarrow_a [\![V]\!]$.

Proof. We prove this theorem by induction on the height of the evaluation derivation. Base cases involve (Abs) and (Labs), and inductive cases use (App) and (Lapp).

(Abs): $\lambda^b x.M_0 \Downarrow_a \lambda^b x.M_0$ where $M = V = \lambda^b x.M_0$. Therefore, $[\![M]\!] \Downarrow_a [\![V]\!]$, which proves this case.

(Tabs): $\Lambda\alpha.V_0 \Downarrow_a \Lambda\alpha.V_0$ where $M = V = \Lambda\alpha.V_0$. Therefore, $[\![M]\!] \Downarrow_a [\![V]\!]$, which proves this case.

(Labs): $\Lambda l.V_0 \Downarrow_a \Lambda l.V_0$ where $M = V = \Lambda l.V_0$. By a careful analysis, it turns out that V_0 is written as $\Lambda l_1.\cdots\Lambda l_n.V_n$ for $n \geq 0$. Let us name $\Lambda l_{i+1}.V_{i+1}$ as V_i for $0 \leq i \leq n-1$, and V_n is either $\lambda^{Loc} x.M_0$ or $\Lambda\alpha.W$. Note that all V_is and V_n must not be a variable since the empty typing environment is in the hypothesis of the typing derivation over M. Hence, we need to prove

$$[\![\Lambda l.\Lambda l_1.\cdots\Lambda l_n.V_n]\!] \Downarrow_a [\![\Lambda l.\Lambda l_1.\cdots\Lambda l_n.V_n]\!]$$

We prove this case by the second induction on the number of the nested location abstractions, n. When $n = 0$, consider $V_0 = \lambda^{Loc} x.M_0$.

$$[\![M]\!] = [\![V]\!] = [\![\Lambda l.\lambda^{Loc} x.M_0]\!] = ([\![(\lambda^{Loc} x.M_0)\{\mathbf{c}/l\}]\!], [\![(\lambda^{Loc} x.M_0)\{\mathbf{s}/l\}]\!])$$

which is translated into $(\lambda^{Loc\{\mathbf{c}/l\}} x.[\![M_0\{\mathbf{c}/l\}]\!], \lambda^{Loc\{\mathbf{s}/l\}} x.[\![M_0\{\mathbf{s}/l\}]\!])$. This translation is well-defined in either case over Loc. If Loc is some b then $Loc\{a/l\}$ is Loc, which is b. If Loc is a location variable, then it must be l since M is a top-level closed term. So, $Loc\{a/l\}$ is a.

By (Abs), (1):$\lambda^{Loc\{\mathbf{c}/l\}} x.[\![M_0\{\mathbf{c}/l\}]\!] \Downarrow_a \lambda^{Loc\{\mathbf{c}/l\}} x.[\![M_0\{\mathbf{c}/l\}]\!]$.

By (Abs), (2):$\lambda^{Loc\{\mathbf{s}/l\}} x.[\![M_0\{\mathbf{s}/l\}]\!] \Downarrow_a \lambda^{Loc\{\mathbf{s}/l\}} x.[\![M_0\{\mathbf{s}/l\}]\!]$.

By (Pair) in the evaluation rule for λ_{rpc} with (1) and (2), $M \Downarrow_a V$, which proves this base case with $V_0 = \lambda^{Loc} x.M_0$ of the second induction.

Also when $n = 0$, consider the other base case with $V_0 = \Lambda\alpha.W$.

$$[\![M]\!] = [\![V]\!] = [\![\Lambda l.\Lambda\alpha.W]\!] = ([\![(\Lambda\alpha.W)\{\mathbf{c}/l\}]\!], [\![(\Lambda\alpha.W)\{\mathbf{s}/l\}]\!])$$

which is translated into $(\Lambda\alpha.[\![W\{\mathbf{c}/l\}]\!], \Lambda\alpha.[\![W\{\mathbf{s}/l\}]\!])$.

By (Tabs), (1)':$\Lambda\alpha.[\![W\{\mathbf{c}/l\}]\!] \Downarrow_a \Lambda\alpha.[\![W\{\mathbf{c}/l\}]\!]$.

By (Tabs), (2)':$\Lambda\alpha.[\![W\{\mathbf{s}/l\}]\!] \Downarrow_a \Lambda\alpha.[\![W\{\mathbf{s}/l\}]\!]$.

By (Pair) with (1)' and (2)', $M \Downarrow_a V$, which proves the remaining base case with $V_0 = \Lambda\alpha.W$ of the second induction.

Now let us prove the inductive case of the second induction, $n > 0$.

$$[\![M]\!] = [\![V]\!] = [\![\Lambda l.\Lambda l_1.V_1]\!] = ([\![(\Lambda l_1.V_1)\{\mathbf{c}/l\}]\!], [\![(\Lambda l_1.V_1)\{\mathbf{s}/l\}]\!])$$

which is $([\![\Lambda l_1.(V_1\{\mathbf{c}/l\})]\!], [\![\Lambda l_1.(V_1\{\mathbf{s}/l\})]\!])$ since l must not be equal to l_1. Then, by I.H. with $n-1$ of the second induction,

$$(3) : [\![\Lambda l_1.(V_1\{\mathbf{c}/l\})]\!] \Downarrow_a [\![\Lambda l_1.(V_1\{\mathbf{c}/l\})]\!] \quad (4) : [\![\Lambda l_1.(V_1\{\mathbf{s}/l\})]\!] \Downarrow_a [\![\Lambda l_1.(V_1\{\mathbf{s}/l\})]\!]$$

By (Pair) in the evaluation rule for λ_{rpc} with (3) and (4), $[\![M]\!] \Downarrow_a [\![V]\!]$, which proves this inductive case of the second induction. Therefore, we prove this case of (Labs).

(App): $L_1 M_1 \Downarrow_a V$, (1):$L_1 \Downarrow_a \lambda^b x.M_0$, (2):$M_1 \Downarrow_a W$, and (3):$M_0\{W/x\} \Downarrow_b V$ where $M = L_1 M_1$. By (T-App), (4):$\emptyset \vdash_a L_1 : B \xrightarrow{b} A$ and (5):$\emptyset \vdash_a M_1 : B$.

By I.H. with (1) and (4), $[\![L_1]\!] \Downarrow_a [\![\lambda^b x.M_0]\!]$, which is (6):$[\![L_1]\!] \Downarrow_a \lambda^b x.[\![M_0]\!]$.

By I.H. with (2) and (5), (7):$[\![M_1]\!] \Downarrow_a W$.

By the type soundness theorem with (1) and (4), (8):$\emptyset \vdash_a \lambda^b x.M_0 : B \xrightarrow{b} A$.

By the type soundness theorem with (2) and (5), (9):$\emptyset \vdash_a W : B$.

By the lemma (Value substitution) with (8) and (9), (10):$\emptyset \vdash_b M_0\{W/x\} : A$.

By I.H. with (3) and (10), (11):$[\![M_0\{W/x\}]\!] \Downarrow_b [\![V]\!]$.

By the lemma (Substitution with the monomorphization translation) with (11), (12):$[\![M_0]\!]\{[\![W]\!]/x\} \Downarrow_b [\![V]\!]$.

By (App) with (6), (7), and (12), $[\![L_1]\!][\![M_1]\!] \Downarrow_a [\![V]\!]$, which proves this case since $[\![L_1 M_1]\!] = [\![L_1]\!][\![M_1]\!]$.

(Tapp): $M_1[B] \Downarrow_a V_0\{B/\alpha\}$, and (1):$M_1 \Downarrow_a \Lambda\alpha.V_0$ where $M = M_1[B]$ and $V = V_0\{B/\alpha\}$.

By (T-Tapp), $\emptyset \vdash_a M_1[B] : A_0\{B/\alpha\}$, (2):$\emptyset \vdash_a M_1 : \forall\alpha.A_0$ where $A = A_0\{B/\alpha\}$.

By I.H. with (1) and (2), $[\![M_1]\!] \Downarrow_a [\![\Lambda\alpha.V_0]\!]$, which is (3):$[\![M_1]\!] \Downarrow_a \Lambda\alpha.[\![V_0]\!]$ by the def. of the translation.

By (Tapp) with (3), $[\![M_1]\!][[\![B]\!]] \Downarrow_a [\![V_0]\!]\{[\![B]\!]/\alpha\}$.

By the lemma 3.9, $[\![M_1]\!][[\![B]\!]] \Downarrow_a [\![V_0\{B/\alpha\}]\!]$, which proves this case.

(Lapp): $L[b] \Downarrow_a V_0\{b/l\}$ and (1):$L \Downarrow_a \Lambda l.V_0$ where $M = L[b]$ and $V = V_0\{b/l\}$. By (T-Lapp), $\emptyset \vdash_a L[b] : A_0\{b/l\}$ and (2):$\emptyset \vdash_L a : \forall l.A_0$ where $A = A_0\{b/l\}$.

By applying I.H. to (1) and (2), $[\![L]\!] \Downarrow_a [\![\Lambda l.V_0]\!]$. By the definition of the translation, (3):$[\![L]\!] \Downarrow_a ([\![V_0\{\mathbf{c}/l\}]\!], [\![V_0\{\mathbf{s}/l\}]\!])$.

Since $[\![L[b]]\!]$ is $\pi_1([\![L]\!])$ if $b = \mathbf{c}$, and it is $\pi_2([\![L]\!])$ if $b = \mathbf{s}$, we prove two sub cases. Suppose $b = \mathbf{c}$. By (Pair) in the evaluation rule for λ_{rpc} with (3), $\pi_1([\![L]\!]) \Downarrow_a V_0\{\mathbf{c}/l\}$, which proves one case of (Lapp) when $b = \mathbf{c}$. Now suppose $b = \mathbf{s}$. By (Pair) with (3) again, $\pi_2([\![L]\!]) \Downarrow_a V_0\{\mathbf{s}/l\}$, which proves the other case of (Lapp).

□

3.3. On putting the polymorphic RPC calculus into practice

Now we have finished presenting our theory of the polymorphic RPC calculus. A strong point is that this is a new RPC calculus supporting polymorphic locations that conservatively extends the typed RPC calculus.

To put this theory into practice, however, there is a weak point to address. Although our theory clearly accounts for what is the polymorphic RPC calculus,

the monomorphization translation, on which the theory depends, can potentially lead to code explosion, as was explained in Section 3.2.1. The reason to perform the monomorphization translation is that after it, one can determine statically if every lambda abstraction is a local or remote procedure call. We call this a *static* approach to the polymorphic RPC calculus.

An alternative way is what we call a *dynamic* approach that allows one to determine dynamically if a given lambda application is a local or remote procedure call in runtime after the slicing compilation. However, in the existing client-server calculi, given a function V and an argument W, three application terms are local procedure call, $V(W)$, a remote procedure call from the client to the server, $\mathsf{req}(V, W)$, and the other remote procedure call for the reverse direction, $\mathsf{call}(V, W)$, and they can be used only where the caller and the callee locations are statically resolved. They are not equipped with such dynamic operation on locations in runtime necessary for lambda applications that are not certain whether they are remote procedure calls. For example, the cross-tier composition function ($\circ$) in Section 3.1.2 gets a typing for a subterm $g\ x$ as

$$\{\alpha\beta\gamma,\ l_1 l_2,\ f : \beta \xrightarrow{l_2} \gamma, g : \alpha \xrightarrow{l_1} \beta, x : \alpha\} \vdash_{l_2} g\ x : \beta$$

where l_1 is where to run the function g and l_2 is where the application is. It cannot be statically determined if $g\ x$ is a remote procedure call or not.

For the dynamic approach to the polymorphic RPC calculus, we need to introduce a new application term that allows such dynamic location checking in runtime as this:

$$\mathsf{gen}(Loc', V, W)$$

where Loc' is the (callee) location of where to run the function V. Note that the caller location, say, Loc, is where this application term runs, which is easy to obtain in runtime. Whenever a lambda application in $\lambda^{\forall}_{rpc}$ poses uncertain caller or callee locations, it can be compiled into this new dynamic application term in a client-server calculus, which we might call a *polymorphic CS calculus*, $\lambda^{\forall}_{cs}$.

The dynamic semantics for $\mathsf{gen}(Loc', f, arg)$ at Loc can be defined as in the following table.

Caller(Loc), Callee(Loc')	$\mathsf{gen}(Loc', f, arg)$	procedure call (flow)
$Loc = Loc' = a$	$f(arg)$	Local (a →a)
$Loc = \mathbf{c}$ and $Loc' = \mathbf{s}$	$req(f, arg)$	Remote ($\mathbf{c}$→ $\mathbf{s}$)
$Loc = \mathbf{s}$ and $Loc' = \mathbf{c}$	$call(f, arg)$	Remote ($\mathbf{s}$→ $\mathbf{c}$)

The first column of the table shows three cases of dynamic checking on caller and callee locations. Note that the caller and the callee locations become monomorphic in runtime. When the caller and callee locations match to one of the three cases, the semantics for the generic application term is defined by the semantics for the matched specific application term in the second column.

Now a slicing compilation may simply compile a lambda application $L\ N$ in $\lambda^{\forall}_{rpc}$ into a generic application term in $\lambda^{\forall}_{cs}$:

$$\mathcal{C}[\![L\ N]\!]_{\Gamma,Loc,B} = \mathsf{gen}(Loc', \mathcal{C}[\![L]\!]_{\Gamma,Loc,A\xrightarrow{Loc'},B}, \mathcal{C}[\![N]\!]_{\Gamma,Loc,A})$$

where $\mathcal{C}[\![M]\!]_{\Gamma,Loc,A}$ denotes a compilation of a term M that has type A at location Loc under a type environment Γ. Then the generic application term will do local or remote procedure calls under the dynamic semantics in $\lambda^{\forall}_{cs}$, as explained previously. For example, in the cross-tier composition function, the subterm $g\ x$ under the typing explained above is compiled into $\mathsf{gen}(l_1, g, x)$ at the location l_2.

Note that whenever the caller and callee locations are determined statically for comparison in compile-time, it is possible to optimize the compilation of lambda applications in $\lambda^{\forall}_{rpc}$ so as to generate more specific application terms in $\lambda^{\forall}_{cs}$ for the purpose of avoiding dynamic location checking.

The dynamic approach could be further optimized in terms of the reduction of dynamically passing locations and checking them if we could make use of the monomorphization translation in the static approach without risking code explosion. Some polymorphic location functions, say, with only one location abstraction such as $\Lambda l.\Lambda\alpha.\lambda^{l}x.x$, had better be monomorphized into a client version and a server version. This is because the polymorphic functions will be placed anyway both in the client and in the server after the slicing compilation.

The design of a *selective* monomorphization translation would be also useful in making the polymorphic RPC calculus into practice. A basic idea is to classify location variables by *kind* indicating if the location variables are static or dynamic. That is, we may replace $\Lambda l.V$ by

$$\Lambda l : k.V \text{ where the kind } k \text{ is either static or dynamic.}$$

For example, two location abstractions in the cross-tier composition function may get kind information as:

$$\Lambda l_1 : \mathit{static}.\ \Lambda l_2 : \mathit{dynamic}.\ \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\lambda^{l_2}f.\lambda^{l_2}g.\lambda^{l_2}x.f\ (g\ x).$$

Then the selective monomorphization translation would be only applied to the polymorphic locations of the static kind.

$$[\![\Lambda l : k.V]\!]_{\forall l:k.A} = \begin{cases} ([\![V\{\mathbf{c}/l\}]\!]_{A\{\mathbf{c}/l\}}, [\![V\{\mathbf{s}/l\}]\!]_{A\{\mathbf{s}/l\}}) & \texttt{if } k = \text{static} \\ \Lambda l : k.[\![V]\!]_A & \texttt{if } k = \text{dynamic} \end{cases}$$

where $[\![M]\!]_A$ denotes the selective monomorphization translation for a term M of type A. When the kind k is static, it is the same as the monomorphization translation. Otherwise, it leaves the location abstraction and translates the body. The remaining dynamic kinded locations after the selective translation would be supported by the polymorphic CS calculus. For example, the location-kinded cross-tier composition function is selectively monomorphized into one instantiated with $\mathbf{c}$ for l_1 and the other with $\mathbf{s}$ for l_1. A benefit is that locations for l_1

can be statically resolved incurring no burden in run-time due to dynamically passing them.

Now a question on the selective monomorphization is how to decide if given location variables are static or dynamic. Generally, there are two options. Programmers may assign location variables appropriate kinds manually, or some static analysis may do this automatically.

While the dynamic approach thus solves the potential code explosion problem of the static approach, it seems to require a *type-passing semantics* where type information is formed and passed to polymorphic functions during run-time. But only the generic application terms depend on location information. Nothing else depends on location nor type information. So, we would like to have a *type-erasure semantics* where types will ultimately be erased and the same term represents different instantiations of polymorphic functions at run-time, resulting in no run-time cost. To retain location information in the type-erasure semantics, we need to introduce *location representations* that can carry run-time location information about location variables and can be type-safely inspected. For this purpose, a generalized algebraic data type (GADT (Xi et al., 2003) or first-class phantom type (Cheney and Hinze, 2003)) can be used. For example *Location* α is such a GADT that has two constructors $\hat{\mathbf{c}}$ of type *Location Client* and $\hat{\mathbf{s}}$ of type *Location Server* where *Client* and *Server* are some types. Then every generic application term $\mathsf{gen}(Loc, M, N)$ in the type-passing semantics can be implemented by a variant as $\mathsf{gen}(L, M, N)$ in the type-erasure semantics where L is a term of *Location* A. The type A will be one of α, *Client*, and *Server* when *Loc* is one of l, $\mathbf{c}$, and $\mathbf{s}$, respectively. The variant generic application terms can be implemented as a case expression, case L of $\{\ \hat{\mathbf{c}} \to M;\ \hat{\mathbf{s}} \to N\ \}$. This shows that using GADT to encode location information fits well for our purpose.

In summary, the polymorphic RPC calculus can be supported either by a static approach or a dynamic approach. In the static approach, no dynamic location checking is required in runtime but there is some potential code explosion problem. In the dynamic approach, no code explosion problem will happen but this advantage comes at the cost of dynamically passing locations and checking them in runtime.

The following table summarizes the status.

RPC calculi	Dynamic check	Location type	Slicing
Untyped RPC	always	(untyped)	yes
Typed RPC	never	mono. loc.	yes
Poly. RPC (Static)	never (code size ↑)	poly. loc.	yes
Poly. RPC (Dynamic)	only for poly loc.	poly. loc.	yes

4. Related work and Discussion

Polymorphic locations. There are only a few publications that are relevant to the notion of polymorphic locations. ML5 supports what they call *world polymorphism* (Murphy et al., 2008; Murphy, 2008). Let us first explain ML5 briefly.

A key construct for remote evaluation is $from\ L\ get\ M$ where L is supposed to evaluate to a world and M is an expression at the world. For example, here is an example program that involves two worlds, home and server:

```
extern bytecode world server
extern val server : server addr @ home
extern val version : unit -> string @ server
extern val alert : string -> unit @ home

fun showversion() =
      let val s = from server
                  get version ()
      in  alert [Server's version is: [s]]
      end

do showversion()
```

where a client at world named *home* invokes a client function *showversion* and, subsequently, it invokes a server function *version* at world named *server* to retrieve a version string and to display it.

In ML5, a function is said to be *valid*, which means it can be used at any world, when it does not access any local resources. An example is *map*, which roughly has type $\Box w.(A \to B) \to ([A] \to [B])$ using the box modal construct[4] over a bound world variable w. Using *map*, we can write as

$$from\ server\ get\ (map\ (fn\ x => x + 1)\ [1, 2, 3]).$$

Although the use of quantified modal construct may look like the use of the universal quantifier over a bound location variable $\forall l$, they are subtly different from each other. Every world variable quantified by a box modal construct is instantiated with some current world. In the example, w becomes instantiated with *server* in the server where *map* is used. The polymorphic RPC calculus expresses location polymorphism by parametric polymorphism. In the example, the client would select a server version of *map* by $map[\mathbf{s}]$ in the client, and would invoke the selected server map function. In this respect, while ML5 can be called modally quantified polymorphism, the polymorphic RPC calculus is based on parametric polymorphism. At least, our approach is believed to be more suitable for the Links programming language. It would be interesting to investigate a formal comparison between world polymorphism and location polymorphism.

For implementation of polymorphism, ML5 represents worlds at run-time to specialize the representation of values given its world while, in (the static approach to) the polymorphic RPC calculus, we compile all location abstractions and applications away during compile-time to know communication flows statically. Here, worlds are treated as concrete addresses of distributed computers while locations are regarded as abstract categories of distributed computers, i.e., client and server. That is why the ML5 example imports the server address as an external value to use it in the client. However, both of ML5 and the polymor-

[4] In fact, it is *sham*, a variant of the box modal construct.

phic RPC calculus can take an advantage of each other's approach. On the one hand, ML5 could enjoy something like our monomorphization translation, for example, as an optimization. On the other hand, the polymorphic RPC calculus could represent locations at run-time after introducing a dynamic operation on locations as in the dynamic approach.

In Eliom (Radanne, 2017; Radanne and Vouillon, 2018), there are a few features to discuss for comparison. First, it supports a macro feature called *shared sections*, which makes it possible to write code for the client and for the server at the same time. It is reported that this technique is pervasively used in Eliom to expose implementations that can be used either on the client or on the server with similar semantics, in a very concise way. But this is a purely syntactic transformation, which is implemented simply by duplicating the code before type checking.

Second, for integration with the OCaml language, Eliom introduces a third location called *base*. Code located on base can be used both on the client and on the server. This feature allows Eliom to be integrated with the OCaml ecosystem smoothly. The type checker reports an error if Eliom programs put on the base location other than OCaml constructs. When a multi-tier programming language is designed to be based on some existing programming language, this feature will be useful.

Third, Eliom allows the same module to mix declarations from multiple locations. Such modules are called *mixed*. This allows programmers to group together declarations that are semantically related, regardless of client-server boundaries. The module type checker enforces an important constraint that as you go down inside sub modules, locations should be properly included. A client module cannot contain server declarations and conversely, but mixed modules can contain everything. It would be interesting to extend the polymorphic RPC calculus with multi-tier ML modules like the ones provided by Eliom.

Location inference. The polymorphic RPC calculus needs a location inference method in practice. Without it, programmers would be burdened because they have to annotate locations on applications as well as on lambda abstractions. It is desirable to have an automatic location inference method.

In the design of a location inference algorithm, an issue is how location abstractions and applications are written in polymorphic RPC programs. A solution is to extend type inference methods for the System F calculus since the inference problem for type abstractions and applications is thought to be similar to one for location counterparts in the polymorphic RPC calculus. Although it is well-known to be an undecidable problem in general (Wells, 1993), there have been some practical trade-offs including (Milner, 1978; Dunfield and Krishnaswami, 2013; Serrano et al., 2018). It would be interesting to see whether their methods can be applied to location inferences successfully.

Besides inference for location constructs, location inference may be designed to insert coercions where there are location conflicts to accept more programs as well-located ones. For example, $\lambda^{\mathbf{c}}(f : A \xrightarrow{\mathbf{s}} B).\ \cdots)\ (\lambda^{\mathbf{c}}x.\ \cdots)$ is ill-typed but the term can be slightly transformed to be well-typed by inserting

an η-conversion coercion over the argument automatically as: $\lambda^{\mathbf{c}}(f : A \xrightarrow{\mathbf{s}} B).\ \cdots)\ (\lambda^{\mathbf{s}} y.(\lambda^{c} x.\ \cdots)\ y)$.

Regarding ML5, it is reported that a simple extension of Hindley-Milner type inference algorithm is developed. The programmer does not usually need to use the box modality manually, because type inference will automatically generalize declarations to be valid whenever possible (Murphy et al., 2008; Murphy, 2008).

Neubauer developed a constraint-based static analysis that statically infers optimal location annotations for operations in the multi-tier calculus (Neubauer, 2007).

Multi-tier calculi. There have been several multi-tier programming languages being developed. Links (Cooper et al., 2007) is a multi-tier web programming language that employs the RPC calculus as the foundation for client-server communication. Lambda5 (Murphy VII et al., 2004; Murphy, 2008) is a modally-typed lambda calculus in which modal type systems can control local resources safely in distributed systems. A multi-tier calculus by Neubauer and Thiemann (Neubauer and Thiemann, 2005) automatically constructs communications for concurrently running processes employing session types (Gay and Hole, 1999) to enforce the integrity of communications. They proposed a series of transformations as compilation to convert a source program into separate programs at different locations determined by the use of primitives that run only at specific locations. There are many other multi-tier web programming languages as follows. Hop (Serrano et al., 2006; Serrano and Berry, 2012) extending Scheme; Hop.js extending JavaScript (Serrano and Prunet, 2016); Eliom, a multi-tier ML programming language, featuring module systems extended with location annotations (Radanne, 2017) in the project Ocsigen (Balat, 2006); and Ur/Web (Chlipala, 2015) with a dependently typed system; a multi-tier functional reactive programming framework, ScalaLoci (Weisenburger et al., 2018), and another interesting framework for Scala (Reynders et al., 2014).

5. Conclusion

This paper proposed the polymorphic RPC calculus where programmers can write succinct multi-tier programs using polymorphic location constructs and the polymorphic multi-tier programs can be automatically translated into monomorphic multi-tier programs only with location constants amenable to the existing slicing compilation methods for client-server model. For the polymorphic RPC calculus, we formulated the type system, and proved its type soundness. Also, we designed the monomorphization translation, and we proved its type and semantic correctness for the translation.

As future work, we want to integrate the polymorphic RPC calculus into the Links programming language (Cooper et al., 2007), which was once designed based on the untyped RPC calculus. But today Links offers many modern features of programming languages on a call-by-value variant of System F with row polymorphism, row-based effect types, and implicit subkinding (Lindley and Cheney, 2012; Hillerström et al., 2017; Fowler et al., 2019). Therefore it

would be a challenge how the polymorphic RPC type system can be integrated smoothly to work with these features.

Another interesting direction is to mechanize the semantics of the polymorphic RPC calculus using a proof assistant to substantiate the proofs in this work and to make the mechanized semantics a basis for further development.

References

Balat, V., 2006. Ocsigen: Typing Web Interaction with Objective Caml, in: Proceedings of the 2006 Workshop on ML (ML '06), pp. 84–94. doi:10.1145/1159876.1159889.

Cheney, J., Hinze, R., 2003. First-Class Phantom Types. Technical Report CUCIS TR2003-1901. Cornell University.

Chlipala, A., 2015. Ur/Web: A Simple Model for Programming the Web. Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15 , 153–165URL: http://dl.acm.org/citation.cfm?doid=2676726.2677004, doi:10.1145/2676726.2677004.

Choi, K., Chang, B., 2019. A theory of RPC calculi for client-server model. Journal of Functional Programming 29, e5. URL: https://doi.org/10.1017/S0956796819000029, doi:10.1017/S0956796819000029.

Cooper, E.K., Lindley, S., Wadler, P., Yallop, J., 2007. Links: Web programming without tiers, in: Proceedings of the 5th International Conference on Formal Methods for Components and Objects, Springer-Verlag, Berlin, Heidelberg. pp. 266–296. URL: http://dl.acm.org/citation.cfm?id=1777707.1777724.

Cooper, E.K., Wadler, P., 2009. The rpc calculus, in: Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, ACM, New York, NY, USA. pp. 231–242. URL: http://doi.acm.org/10.1145/1599410.1599439, doi:10.1145/1599410.1599439.

Dunfield, J., Krishnaswami, N.R., 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism, in: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA. pp. 429–442. URL: http://doi.acm.org/10.1145/2500365.2500582, doi:10.1145/2500365.2500582.

Fowler, S., Lindley, S., Morris, J.G., Decova, S., 2019. Exceptional asynchronous session types: Session types without tiers. Proc. ACM Program. Lang. 3, 28:1–28:29. URL: http://doi.acm.org/10.1145/3290341, doi:10.1145/3290341.

Gay, S.J., Hole, M., 1999. Types and subtypes for client-server interactions, in: Proceedings of the 8th European Symposium on Programming Languages and Systems, Springer-Verlag, London, UK, UK. pp. 74–90. URL: http://dl.acm.org/citation.cfm?id=645393.756448.

Harper, R., Morrisett, G., 1995. Compiling polymorphism using intensional type analysis, in: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA. pp. 130–141. URL: http://doi.acm.org/10.1145/199448.199475, doi:10.1145/199448.199475.

Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.C., 2017. Continuation Passing Style for Effect Handlers, in: Miller, D. (Ed.), 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 18:1–18:19. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7739, doi:10.4230/LIPIcs.FSCD.2017.18.

Lindley, S., Cheney, J., 2012. Row-based effect types for database integration, in: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, ACM, New York, NY, USA. pp. 91–102. URL: http://doi.acm.org/10.1145/2103786.2103798, doi:10.1145/2103786.2103798.

Milner, R., 1978. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348 – 375. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144, doi:https://doi.org/10.1016/0022-0000(78)90014-4.

Murphy, VII., T., 2008. Modal Types for Mobile Code. Ph.D. thesis. Carnegie Mellon University. Pittsburgh, PA, USA. AAI3314655.

Murphy, VII., T., Crary, K., Harper, R., 2008. Type-safe distributed programming with ml5, in: Proceedings of the 3rd Conference on Trustworthy Global Computing, Springer-Verlag, Berlin, Heidelberg. pp. 108–123. URL: http://dl.acm.org/citation.cfm?id=1793574.1793585.

Murphy VII, T., Crary, K., Harper, R., Pfenning, F., 2004. A symmetric modal lambda calculus for distributed computing, in: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Washington, DC, USA. pp. 286–295. URL: http://dx.doi.org/10.1109/LICS.2004.7, doi:10.1109/LICS.2004.7.

Neubauer, M., 2007. Multi-tier programming. Ph.D. thesis. Universitt Freiburg.

Neubauer, M., Thiemann, P., 2005. From sequential programs to multi-tier applications by program transformation, in: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA. pp. 221–232. URL: http://doi.acm.org/10.1145/1040305.1040324, doi:10.1145/1040305.1040324.

Radanne, G., 2017. Tierless Web programming in ML. Ph.D. thesis. University Paris Diderot.

Radanne, G., Vouillon, J., 2018. Tierless web programming in the large, in: Companion Proceedings of the The Web Conference 2018, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland. pp. 681–689. URL: https://doi.org/10.1145/3184558.3185953, doi:10.1145/3184558.3185953.

Reynders, B., Devriese, D., Piessens, F., 2014. Multi-Tier Functional Reactive Programming for the Web, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '14, pp. 55–68. URL: http://dl.acm.org/citation.cfm?doid=2661136.2661140, doi:10.1145/2661136.2661140.

Serrano, A., Hage, J., Vytiniotis, D., Peyton Jones, S., 2018. Guarded impredicative polymorphism, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA. pp. 783–796. URL: http://doi.acm.org/10.1145/3192366.3192389, doi:10.1145/3192366.3192389.

Serrano, M., Berry, G., 2012. Multitier Programming in Hop. Commun. ACM 55, 53–59. URL: http://doi.acm.org/10.1145/2240236.2240253, doi:10.1145/2240236.2240253.

Serrano, M., Gallesio, E., Loitsch, F., 2006. Hop: a Language for Programming the Web 2.0, in: Proceedings of the 1st Dynamic Languages Symposium, Portland, OR, USA. pp. 975–985. doi:10.1145/1176617.1176756.

Serrano, M., Prunet, V., 2016. A Glimpse of Hopjs. SIGPLAN Not. 51, 180–192. URL: http://doi.acm.org/10.1145/3022670.2951916, doi:10.1145/3022670.2951916.

Weisenburger, P., Köhler, M., Salvaneschi, G., 2018. Distributed system development with scalaloci. Proc. ACM Program. Lang. 2, 129:1–129:30. URL: http://doi.acm.org/10.1145/3276499, doi:10.1145/3276499.

Wells, J., B., 1993. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. URL: https://open.bu.edu/handle/2144/1474.

Xi, H., Chen, C., Chen, G., 2003. Guarded recursive datatype constructors. SIGPLAN Not. 38, 224235. URL: https://doi.org/10.1145/640128.604150, doi:10.1145/640128.604150.

Appendix A. Proofs of Lemmas in Section 3.1 A polymorphic RPC calculus

A.1. Proofs of Lemmas in Section 3.1.1 Type soundness

Lemma 3.1 (Value relocation)**.** *If* $\Gamma \vdash_{Loc} V : A$ *then* $\Gamma \vdash_{Loc'} V : A$.

Proof. This lemma is proved by induction on the height of the typing derivation tree in the condition. Base cases use (T-Var) and (T-Abs). Let V be x where (T-Var) is used. Then we have $\Gamma(x) = A$. By (T-Var) with Loc' this time, $\Gamma \vdash_{Loc'} x : A$. This proves one base case with (T-Var). For the other base case, the lemma is provable similarly.

Inductive cases involve (T-Tabs) and (T-Labs). Let V be $\Lambda\alpha.V_0$ and A be $\forall\alpha.A_0$ where (T-Tabs) is used. The condition of the lemma gives us the subtree with (1):$\Gamma, \alpha \vdash_{Loc} V_0 : A_0$.

By applying the induction hypothesis to (1), we will have (2):$\Gamma, \alpha \vdash_{Loc'} V_0 : A_0$.

By applying (T-Tabs) to (2), we can derive the typing derivation tree in the conclusion of the lemma: $\Gamma \vdash_{Loc'} \Lambda\alpha.V_0 : \forall\alpha.A_0$.

For the other inductive case, the lemma can be proved in a similar way. □

Lemma 3.2 (Value substitution)**.** *If* $\Gamma \vdash_{Loc} \lambda^{Loc'} x.M : A \xrightarrow{Loc'} B$ *and* $\Gamma \vdash_{Loc} V : A$ *then* $\Gamma \vdash_{Loc'} M\{V/x\} : B$.

Proof. By (T-Abs) with the first part of the condition, $\Gamma, x : A \vdash_{Loc'} M : B$. Since x is a bound variable, x cannot appear as a free variable in V. By the lemma (Value relocation), we have $\Gamma \vdash_{Loc'} V : A$ from the second part.

Then we have only to show a generalized lemma as: if (1):$\Gamma, x : A \vdash_{Loc'} M : B$, (2):$\Gamma \vdash_{Loc'} V : A$, and (3):$x \notin fv(V)$ then (4):$\Gamma \vdash_{Loc'} M\{V/x\} : B$.

We prove the generalized lemma by induction on the height of the derivation tree (1). The only base case uses (T-Var) with $M = y$. We do case analysis by $y = x$ and $y \neq x$. In either cases, the generalized lemma is provable immediately.

For inductive cases, let us first consider a case using (T-App) with $M = L\ N$. The instance of (1) becomes $\Gamma, x : A \vdash_{Loc'} L\ N : B$.

By (T-App), we have (5):$\Gamma, x : A \vdash_{Loc'} L : C \xrightarrow{Loc''} B$ and (6):$\Gamma, x : A \vdash_{Loc'} N : C$.

By I.H. with (5), (7):$\Gamma \vdash_{Loc'} L\{V/x\} : C \xrightarrow{Loc''} B$.

By I.H. with (6), (8):$\Gamma \vdash_{Loc'} N\{V/x\} : C$.

By (T-App) with (7) and (8), $\Gamma \vdash_{Loc'} (L\{V/x\})\ (N\{V/x\}) : B$ where $(L\{V/x\})\ (N\{V/x\})$ is $(L\ N)\{V/x\}$. Hence, the case is proved.

The other inductive cases use (T-Abs), (T-Tabs), (T-Tapp), (T-Labs), and (T-Lapp), which are all provable similarly. □

Lemma 3.3 (Type substitution)**.** *If* $\Gamma \vdash_{Loc} \Lambda\alpha.V : \forall\alpha.A$ *then* $\Gamma \vdash_{Loc} V\{B/\alpha\} : A\{B/\alpha\}$.

Proof. By (T-Tabs) with the first part of the condition, $\Gamma, \alpha \vdash_{Loc} V : A$. Since α is a bound type variable, α cannot occur in Γ, i.e., $\alpha \notin ftv(\Gamma)$.

We prove a generalized lemma as: if (1):$\Gamma, \alpha \vdash_{Loc} M : A$ and (2):$\alpha \notin ftv(\Gamma)$ then (3):$\Gamma \vdash_{Loc} M\{B/\alpha\} : A\{B/\alpha\}$.

This generalized lemma is proved by induction on the height of the derivation tree (1). The base case uses (T-Var) with $M = x$. $M\{B/\alpha\} = x\{B/\alpha\} = x$. $A\{B/\alpha\}$ must be A. Otherwise, α occurs in A for $\Gamma(x) = A$, which violates (2).

For inductive cases, consider a case using (T-Tapp) with $M = L[C]$ and $A = A_0\{C/\beta\}$. (1) becomes (4):$\Gamma, \alpha \vdash_{Loc} L[C] : A_0\{C/\beta\}$. By (T-Tapp) with (4), we have (5):$\Gamma, \alpha \vdash_{Loc} L : \forall\beta.A_0$. By I.H. with (5), (6):$\Gamma \vdash_{Loc} L\{B/\alpha\} : (\forall\beta.A_0)\{B/\alpha\}$. Note that $(\forall\beta.A_0)\{B/\alpha\} = \forall\beta.(A_0\{B/\alpha\})$ since $\alpha \neq \beta$. By applying (T-Tapp) to (6) with $C\{B/\alpha\}$, (7):$\Gamma \vdash_{Loc} L\{B/\alpha\}[C\{B/\alpha\}] : (A_0\{B/\alpha\})\{C\{B/\alpha\}/\beta\}$. This proves the inductive case by $\Gamma \vdash_{Loc} (L[C])\{B/\alpha\} : (A_0\{C/\beta\})\{B/\alpha\}$. The other inductive cases use (T-Abs), (T-App), (T-Tabs), (T-Labs), and (T-Lapp), which are provable similarly. □

Lemma 3.4 (Location substitution)**.** *If* $\Gamma \vdash_{Loc} \Lambda l.V : \forall l.A$ *then* $\Gamma \vdash_{Loc} V\{Loc'/l\} : A\{Loc'/l\}$.

Proof. By (T-Labs) with the first part of the condition, $\Gamma, l \vdash_{Loc} V : A$. Since l is a bound location variable, l cannot occur in Γ, i.e., $l \notin flv(\Gamma)$.

We prove a generalized lemma as: if (1):$\Gamma, l \vdash_{Loc} M : A$ and (2):$l \notin flv(\Gamma)$ then (3):$\Gamma \vdash_{Loc} M\{Loc'/l\} : A\{Loc'/l\}$. In the base case, $M = x$. $M\{Loc'/l\} = x\{Loc'/l\} = x$. $A\{Loc'/l\} = A$ because of $\Gamma(x) = A$ and (2).

For inductive cases, let us first a case using (T-Lapp) with $M = L[Loc_0]$ and $A = B\{Loc_0/l_0\}$. The instance of (1) becomes (4):$\Gamma, l \vdash_{Loc} L[Loc_0] : B\{Loc_0/l_0\}$.

By (T-Lapp) with (4), we have (5):$\Gamma, l \vdash_{Loc} L : \forall l_0.B$.

By I.H. with (5), (6):$\Gamma \vdash_{Loc} L\{Loc'/l\} : (\forall l_0.B)\{Loc'/l\}$. Since $l \neq l_0$, (7):$(\forall l_0.B)\{Loc'/l\} = \forall l_0.(B\{Loc'/l\})$.

By (T-Lapp) with (6), (7), and $Loc_0\{Loc'/l\}$, we can derive

$$\Gamma \vdash_{Loc} (L\{Loc'/l\})[Loc_0\{Loc'/l\}] : (B\{Loc'/l\})\{Loc_0\{Loc'/l\}/l_0\}$$

which is $\Gamma \vdash_{Loc} (L[Loc_0])\{Loc'/l\} : (B\{Loc_0/l_0\})\{Loc'/l\}$.

The other inductive cases use (T-Abs), (T-App), (T-Tabs), (T-Tapp), and (T-Labs), which are proved similarly. □

Appendix B. Proofs of Lemmas in Section 3.2 A monomorphization translation of the polymorphic RPC calculus

B.1. Proofs of Lemmas in Section 3.2.2 Type correctness

Lemma 3.5 (Type substitution over type under monomorphization)**.** $[\![A]\!]\{[\![B]\!]/\alpha\} = [\![A\{B/\alpha\}]\!]$.

Proof. We prove this lemma by the structural induction on A. We have two base cases. Let us first consider when $A = base$. In the left-hand side of the equation, $[\![base]\!]\{[\![B]\!]/\alpha\} = base\{[\![B]\!]/\alpha\} = base$. In the right-hand side, $[\![base\{B/\alpha\}]\!] = base$.

For the other base case that A is a type variable, say, β, we do a case analysis on if β is the same as α or not. When $\beta \neq \alpha$, this can be proved similarly as for the first base case. When $\beta = \alpha$, $[\![\alpha]\!]\{[\![B]\!]/\alpha\} = \alpha\{[\![B]\!]/\alpha\} = [\![B]\!] = [\![\alpha\{B/\alpha\}]\!]$.

There are three inductive cases where A is a function type, a polymorphic type, and a polymorphic location.

Case A **is** $A_1 \xrightarrow{a} A_2$**:**

$$\begin{array}{lll} & [\![A_1 \xrightarrow{a} A_2]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & ([\![A_1]\!] \xrightarrow{a} [\![A_2]\!])\{[\![B]\!]/\alpha\} & \\ = & [\![A_1]\!]\{[\![B]\!]/\alpha\} \xrightarrow{a} [\![A_2]\!]\{[\![B]\!]/\alpha\} & \text{by I.H.} \\ = & [\![A_1\{B/\alpha\}]\!] \xrightarrow{a} [\![A_2\{B/\alpha\}]\!] & \text{by def. of } [\![-]\!] \\ = & [\![A_1\{B/\alpha\} \xrightarrow{a} A_2\{B/\alpha\}]\!] & \\ = & [\![(A_1 \xrightarrow{a} A_2)\{B/\alpha\}]\!] & \end{array}$$

Case A **is** $\forall\beta.A_0$**:**

i) $\beta = \alpha$

$$\begin{array}{lll} & [\![\forall\beta.A_0]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & (\forall\beta.[\![A_0]\!])\{[\![B]\!]/\alpha\} & \\ = & \forall\beta.[\![A_0]\!] & \text{by def. of } [\![-]\!] \\ = & [\![\forall\beta.A_0]\!] & \\ = & [\![(\forall\beta.A_0)\{B/\alpha\}]\!] & \end{array}$$

ii) $\beta \neq \alpha$

$$\begin{array}{lll} & [\![\forall\beta.A_0]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & (\forall\beta.[\![A_0]\!])\{[\![B]\!]/\alpha\} & \\ = & \forall\beta.([\![A_0]\!]\{[\![B]\!]/\alpha\}) & \text{by I.H.} \\ = & \forall\beta.([\![A_0\{B/\alpha\}]\!]) & \\ = & [\![\forall\beta.(A_0\{B/\alpha\})]\!] & \\ = & [\![(\forall\beta.A_0)\{B/\alpha\}]\!] & \end{array}$$

Case A **is** $\forall l.A_0$**:**

$$\begin{array}{lll} & [\![\forall l.A_0]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & ([\![A_0\{\mathbf{c}/l\}]\!] \times [\![A_0\{\mathbf{s}/l\}]\!])\{[\![B]\!]/\alpha\} & \\ = & ([\![A_0\{\mathbf{c}/l\}]\!] \times [\![A_0\{\mathbf{s}/l\}]\!])\{[\![B]\!]/\alpha\} & \\ = & [\![A_0\{\mathbf{c}/l\}]\!]\{[\![B]\!]/\alpha\} \times [\![A_0\{\mathbf{s}/l\}\{[\![B]\!]/\alpha\}]\!] & \text{by I.H.} \\ = & [\![(A_0\{\mathbf{c}/l\})\{B/\alpha\}]\!] \times [\![(A_0\{\mathbf{s}/l\})\{B/\alpha\}]\!] & \\ = & [\![(A_0\{B/\alpha\})\{\mathbf{c}/l\}]\!] \times [\![(A_0\{B/\alpha\})\{\mathbf{s}/l\}]\!] & \text{by def. of } [\![-]\!] \\ = & [\![\forall l.(A_0\{B/\alpha\})]\!] & \\ = & [\![(\forall l.A_0)\{B/\alpha\}]\!] & \end{array}$$

□

Lemma 3.6 (Location polymorphism)**.** *Suppose* $\Gamma = \{l_1, \cdots, l_n\} \cup \Gamma_0$ *such that* Γ_0 *has no location variables. If* $\Gamma \vdash_{Loc} M : A$ *then* $(\Gamma_0 \vdash_{Loc} M : A)\{a_1/l_1, \cdots, a_n/l_n\}$ *for any* $a_1, \cdots, a_n$ *with the same height.*

Proof. The proof can be done straightforwardly by induction on the height of the derivation tree for the typing judgment in the condition. □

B.2. Proofs of Lemmas in Section 3.2.3 Semantics correctness

Lemma 3.8 (Substitution under monomorphization)**.** $[\![M]\!]\{[\![W]\!]/x\} = [\![M\{W/x\}]\!]$.

Proof. We prove this lemma by the structural induction on M. In the base case, $M = y$. When $y = x$, $[\![x]\!]\{[\![W]\!]/x\} = x\{[\![W]\!]/x\} = [\![W]\!] = [\![x\{W/x\}]\!]$. When $y \neq x$, $[\![y]\!]\{[\![W]\!]/x\} = y\{[\![W]\!]/x\} = y = [\![y]\!] = [\![y\{W/x\}]\!]$.

For the inductive cases, the proof is done as follows.

Case M **is** $\lambda^a y.L$**:**

i) $y = x$

$$
\begin{array}{lll}
 & [\![\lambda^a y.L]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & (\lambda^a y.[\![L]\!])\{[\![W]\!]/x\} & \\
= & \lambda^a y.[\![L]\!] & \\
= & [\![\lambda^a y.L]\!] & \\
= & [\![(\lambda^a y.L)\{W/x\}]\!] &
\end{array}
$$

ii) $y \neq x$

$$
\begin{array}{lll}
 & [\![\lambda^a y.L]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & (\lambda^a y.[\![L]\!])\{[\![W]\!]/x\} & \\
= & \lambda^a y.([\![L]\!]\{[\![W]\!]/x\}) & \text{by I.H.} \\
= & \lambda^a y.[\![L\{W/x\}]\!] & \text{by def. of } [\![-]\!] \\
= & [\![\lambda^a y.(L\{W/x\})]\!] & \\
= & [\![(\lambda^a y.L)\{W/x\}]\!] &
\end{array}
$$

Case M **is** LN**:**

$$
\begin{array}{lll}
 & [\![LN]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & ([\![L]\!][\![N]\!])\{[\![W]\!]/x\} & \\
= & ([\![L]\!]\{[\![W]\!]/x\})([\![N]\!]\{[\![W]\!]/x\}) & \text{by I.H.} \\
= & ([\![L\{W/x\}]\!])([\![N\{W/x\}]\!]) & \\
= & [\![(L\{W/x\})(N\{W/x\}]\!]) & \\
= & [\![(LN)\{W/x\}]\!] &
\end{array}
$$

Case M **is** $\Lambda l.V$**:**

$$
\begin{array}{lll}
 & [\![\Lambda l.V]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & ([\![V\{\mathbf{c}/l\}]\!], [\![V\{\mathbf{s}/l\}]\!])\{[\![W]\!]/x\} & \\
= & ([\![V\{\mathbf{c}/l\}]\!]\{[\![W]\!]/x\}, [\![V\{\mathbf{s}/l\}]\!]\{[\![W]\!]/x\}) & \text{by I.H.} \\
= & ([\![(V\{\mathbf{c}/l\})\{W/x\}]\!], [\![(V\{\mathbf{s}/l\})\{W/x\}]\!] & \\
= & ([\![(V\{W/x\})\{\mathbf{c}/l\}]\!], [\![(V\{W/x\})\{\mathbf{s}/l\}]\!] & \text{by def. of } [\![-]\!] \\
= & ([\![\Lambda l.(V\{W/x\})]\!] & \\
= & ([\![(\Lambda l.V)\{W/x\}]\!] &
\end{array}
$$

Case M **is** $\Lambda\alpha.V$**:**

$$
\begin{array}{lll}
 & [\![\Lambda\alpha.V]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & (\Lambda\alpha.[\![V]\!])\{[\![W]\!]/x\} & \\
= & \Lambda\alpha.([\![V]\!]\{[\![W]\!]/x\}) & \text{by I.H.} \\
= & \Lambda\alpha.([\![V\{W/x\}]\!]) & \text{by def. of } [\![-]\!] \\
= & [\![\Lambda\alpha.(V\{W/x\})]\!] & \\
= & [\![(\Lambda\alpha.V)\{W/x\}]\!] &
\end{array}
$$

Case M **is** $L[a]$**:**

$$
\begin{array}{lll}
 & [\![L[a]]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & (\pi_i[\![L]\!])\{[\![W]\!]/x\} & i=1 \text{ if } a=\mathbf{c}, \quad i=2 \text{ if } a=\mathbf{s} \\
= & \pi_i([\![L]\!]\{[\![W]\!]/x\}) & \text{by I.H.} \\
= & \pi_i([\![L\{W/x\}]\!]) & \\
= & [\![L\{W/x\}[a]]\!] & \text{by def. of } [\![-]\!] \\
= & [\![(L[a])\{W/x\}]\!] &
\end{array}
$$

Case M **is** $L[B]$**:**

$$
\begin{array}{lll}
 & [\![L[B]]\!]\{[\![W]\!]/x\} & \text{by def. of } [\![-]\!] \\
= & [\![L]\!][[\![B]]\!]\{[\![W]\!]/x\} & \\
= & ([\![L]\!]\{[\![W]\!]/x\})[[\![B]]\!] & \text{by I.H.} \\
= & [\![L\{W/x\}]\!][[\![B]]\!] & \text{by def. of } [\![-]\!] \\
= & [\![L\{W/x\}[B]]\!] & \\
= & [\![(L[B])\{W/x\}]\!] &
\end{array}
$$

□

Lemma 3.9 (Type substitution over term under monomorphization)**.** $[\![V]\!]\{[\![B]\!]/\alpha\} = [\![V\{B/\alpha\}]\!]$.

Proof. We prove a slightly general lemma as $[\![M]\!]\{[\![B]\!]/\alpha\} = [\![M\{B/\alpha\}]\!]$. We prove the general lemma by the structural induction on M. For the base case $M = x$, $[\![x]\!]\{[\![B]\!]/\alpha\} = x\{[\![B]\!]/\alpha\} = x = [\![x]\!] = [\![x\{B/\alpha\}]\!]$.

For the inductive cases, the proof is done as follows.

Case M **is** $\lambda^a x.L$**:**

$$
\begin{array}{lll}
 & [\![\lambda^a x.L]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\
= & (\lambda^a x.[\![L]\!])\{[\![B]\!]/\alpha\} & \\
= & \lambda^a x.([\![L]\!]\{[\![B]\!]/\alpha\}) & \text{by I.H.} \\
= & \lambda^a x.[\![L\{B/\alpha\}]\!] & \\
= & [\![\lambda^a x.(L\{B/\alpha\})]\!] & \text{by def. of } [\![-]\!] \\
= & [\![(\lambda^a x.L)\{B/\alpha\}]\!] &
\end{array}
$$

Case M **is** $\Lambda l.V$**:**

$$
\begin{array}{lll}
 & [\![\Lambda l.V]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\
= & ([\![V\{\mathbf{c}/l\}]\!], [\![V\{\mathbf{s}/l\}]\!])\{[\![B]\!]/\alpha\} & \\
= & ([\![V\{\mathbf{c}/l\}]\!]\{[\![B]\!]/\alpha\}, [\![V\{\mathbf{s}/l\}]\!]\{[\![B]\!]/\alpha\}) & \text{by applying I.H. twice} \\
= & ([\![(V\{\mathbf{c}/l\})\{B/\alpha\}]\!], [\![(V\{\mathbf{s}/l\})\{B/\alpha\}]\!] & \\
= & ([\![(V\{B/\alpha\})\{\mathbf{c}/l\}]\!], [\![(V\{B/\alpha\})\{\mathbf{s}/l\}]\!] & \text{by def. of } [\![-]\!] \\
= & [\![\Lambda l.(V\{B/\alpha\})]\!] & \\
= & [\![(\Lambda l.V)\{B/\alpha\}]\!] &
\end{array}
$$

Case M **is** LN**:**

$$
\begin{array}{lll}
 & [\![LN]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\
= & ([\![L]\!][\![N]\!])\{[\![B]\!]/\alpha\} & \\
= & ([\![L]\!]\{[\![B]\!]/\alpha\}([\![N]\!]\{[\![B]\!]/\alpha\}) & \text{by applying I.H. twice} \\
= & [\![L\{B/\alpha\}]\!][\![N\{B/\alpha\}]\!] & \text{by def. of } [\![-]\!] \\
= & [\![(L\{B/\alpha\})(N\{B/\alpha\})]\!] & \\
= & [\![(LN)\{B/\alpha\}]\!] &
\end{array}
$$

Case M **is** $L[a]$**:**

$$\begin{array}{lll} & [\![L[a]]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \quad (i = 1 \text{ if } a = \mathbf{c},\ i = 2 \text{ if } a = \mathbf{s}) \\ = & (\pi_i[\![L]\!])\{[\![B]\!]/\alpha\} & \\ = & \pi_i([\![L]\!]\{[\![B]\!]/\alpha\}) & \text{by I.H.} \\ = & \pi_i[\![L\{B/\alpha\}]\!] & \text{by def. of } [\![-]\!] \\ = & [\![(L\{B/\alpha\})[a]]\!] & \\ = & [\![(L[a])\{B/\alpha\}]\!] & \end{array}$$

Case M **is** $\Lambda\beta.V$**:**

i) $\beta = \alpha$

$$\begin{array}{lll} & [\![\Lambda\alpha.V]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & (\Lambda\alpha.[\![V]\!])\{[\![B]\!]/\alpha\} & \\ = & \Lambda\alpha.[\![V]\!] & \text{by def. of } [\![-]\!] \\ = & [\![\Lambda\alpha.V]\!] & \\ = & [\![(\Lambda\alpha.V)\{B/\alpha\}]\!] & \end{array}$$

i) $\beta \neq \alpha$

$$\begin{array}{lll} & [\![\Lambda\beta.V]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & (\Lambda\beta.[\![V]\!])\{[\![B]\!]/\alpha\} & \\ = & \Lambda\beta.([\![V]\!]\{[\![B]\!]/\alpha\}) & \text{by I.H.} \\ = & \Lambda\beta.([\![V\{B/\alpha\}]\!]) & \\ = & [\![\Lambda\beta.(V\{B/\alpha\})]\!] & \\ = & [\![(\Lambda\beta.V)\{B/\alpha\}]\!] & \end{array}$$

Case M **is** $L[A]$**:**

$$\begin{array}{lll} & [\![L[A]]\!]\{[\![B]\!]/\alpha\} & \text{by def. of } [\![-]\!] \\ = & ([\![L]\!][[\![A]\!]])\{[\![B]\!]/\alpha\} & \\ = & ([\![L]\!]\{[\![B]\!]/\alpha\})[[\![A]\!]\{[\![B]\!]/\alpha\}] & \text{by I.H. and Lemma 3.5} \\ = & [\![L\{B/\alpha\}]\!][[\![A\{B/\alpha\}]\!]] & \text{by def. of } [\![-]\!] \\ = & [\![L\{B/\alpha\}[A\{B/\alpha\}]]\!] & \\ = & [\![(L[A])\{B/\alpha\}]\!] & \end{array}$$

□