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 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...
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:

```links
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. `main` 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 \rightarrow String` while `getCredentials` has type `String \rightarrow 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].

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 supported to translate them into location-specific versions, for example, one written in JavaScript for the client and the other written in O’Caml for the server, automatically. It is much like the convenience of polymorphic 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, 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, and to instantiate them by location application. For example, the location-neutral map function can be written in Links as ‘fun map(f, xs) l { the body of map }’ where the location attribute is replaced with a location variable l. In other words,

\[ map = Λl.λf.λl.\text{the body of map} \]

that has type \( ∀l.(A \xrightarrow{l} B) \xrightarrow{l} ([A] \xrightarrow{l} [B]) \). The type of \( f \) is \( A \xrightarrow{l} B \), the type of \( xs \) 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 \( map[c] \), which becomes \( λf.λxs. \cdots \) of type \( (A \xrightarrow{c} B) \xrightarrow{c} ([A] \xrightarrow{c} [B]) \) by replacing all
occurrences of \( l \) with \( c \). To run it as a server function, \( map[s] \) will be used. Thus, every polymorphic \( \lambda \)-abstraction can be regarded as a location-neutral one, and a choice of a location specific \( \lambda \)-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 reminds readers of 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 works in Section 4, we conclude in Section 5.

2. Background: The typed RPC calculus

In this section, we remind the reader of the typed RPC calculus. Figure 2 shows a typed RPC calculus \( \lambda_{rpc} \). It is a call-by-value \( \lambda \)-calculus with location annotations on \( \lambda \)-abstractions. The location annotations tell where the \( \lambda \)-abstractions must execute. The client-server model is assumed in the design of the calculus, and so the location annotations are either \( c \) denoting client or \( s \) denoting server.
Syntax

Location  \( a, b \) ::= c | s

Term  \( L, M, N \) ::= V | L M | M[A]

Value  \( V, W \) ::= x | \( \lambda a.x.M \) | \( \Lambda \alpha.V \)

Semantics

\[
\begin{align*}
\lambda^a.x.M \Downarrow_a \lambda^a.x.M \\
L \Downarrow_a \lambda^b.x.N \\
M \Downarrow_a W \\
N\{W/x\} \Downarrow_b V \\
\end{align*}
\]

(App)

\[
\begin{align*}
\Lambda \alpha.V \Downarrow_a \Lambda \alpha.V \\
M \Downarrow_a \Lambda \alpha.V \\
M[B] \Downarrow_a V\{B/\alpha\} \\
\end{align*}
\]

(Tapp)

Figure 2: A typed RPC calculus \( \lambda_{rpc} \)

denoting server. The syntax of the typed RPC calculus is thus defined as shown in the figure.

The semantics of \( \lambda_{rpc} \) 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 \( \lambda \)-abstraction and itself. (App) is more interesting for an application \( L M \) at location \( a \): it performs \( \beta \)-reduction at location \( b \), which a \( \lambda \)-abstraction from \( L \) has as an annotation, with a value \( W \) from \( M \), and it continues to evaluate the \( \beta \)-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 \( a \) is different from \( b \). 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 \( \lambda \)-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 \( \alpha \) 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 \( \alpha \) in \( V \).

A type system for the typed RPC calculus in Figure 2 basically comes from the one in (Choi and Chang, 2019) that accounts for remote procedure calls in 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 \( \alpha \) in \( A \). Accordingly, every typing environment \( \Gamma \) now has type variables as well as associations of variables and types in general as \( \{ \alpha_1, \ldots, \alpha_k, x_1 : A_1, \ldots, x_m : A_m \} \).
Types

\[ Type \ A, B, C \ ::= \ base \mid A \rightarrow A \mid \alpha \mid \forall \alpha.A \]

Typing Rules

\[
\begin{align*}
(T-Var) & \quad \Gamma(x) = A & \Gamma \vdash_a x : A \\
(T-Abs) & \quad \Gamma, x : A \vdash_b M : B & \Gamma \vdash_a \lambda^b x. M : A \rightarrow B \\
(T-App) & \quad \Gamma \vdash_a L : A \rightarrow B \quad \Gamma \vdash_a M : A & \Gamma \vdash_a LM : B \\
(T-Tabs) & \quad \Gamma,\alpha \vdash_a V : A & \Gamma \vdash_a \Lambda^\alpha \alpha.V : \forall \alpha.A \\
(T-Tapp) & \quad \Gamma \vdash_a M : \forall \alpha.A & \Gamma \vdash_a M[B] : A\{B/\alpha\}
\end{align*}
\]

Figure 3: A type system for the typed RPC calculus

\[ A_m \}. \]

The type system has two features related to location. First, location annotations are introduced to function types as \( A \rightarrow_a B \). Every \( \lambda \)-abstraction that must run at location \( a \) gets this function type. For example, \( (\lambda s.f. (f M)) (\lambda c y. \cdots) \) is well-typed when \( f \) is of type \( A \rightarrow_c B \) for some types \( A \) and \( B \). However, \( (\lambda^c f. \cdots (\lambda^h h. \cdots) f \cdots (\lambda^g y. \cdots) f \cdots) \) is ill-typed when \( h \) is of \( A \rightarrow_c B \) and \( g \) is of \( A \rightarrow_s B \) since the type language of the typed RPC calculus only allows either \( c \) or \( s \), 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 \( \Gamma \). (T-Var) is defined as usual. (T-Abs) assigns \( \lambda \)-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 \( \lambda \)-abstraction.

Combining these two features, (T-App) is designed to be a refinement of the conventional \( \lambda \)-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 \( b \), \( LM \) is statically found to be a remote procedure call: if \( a = c \) and \( b = s \), it is to invoke a server function from the client, and if \( a = s \) and \( b = c \), 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 \( \lambdarpc \) into a client program and a server program in \( \lambda^{enc}_{cs} \) or \( \lambda^{state}_{cs} \), 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 \texttt{req(V,W)} is used to invoke a server function \texttt{V} with an argument \texttt{W} from the client, and \texttt{call(V,W)} is another remote procedure call construct in the reverse direction. In separate programs, we use \texttt{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 location function and \(W\) is an argument. When \(a = c\) and \(b = s\), the slicing compilation methods generate \texttt{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 \(a = s\) and \(b = c\), the slicing compilation methods generate \texttt{call(V,W)} where \(V\) is a client function. This term is implemented in a similar way for \texttt{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 \(\lambda^{enc}_{cs}\) or \(\lambda^{state}_{cs}\) 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.

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.

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 \(a\). Accordingly, a new
Syntax

| Location | a, b |
|-----------|------|
|           | c    |
|           | s    |

| Term      | L, M, N |
|-----------|---------|
|           | V      |
|           | L M    |
|           | M[A]   |
|           | M[Loc] |

| Value     | V, W    |
|-----------|---------|
|           | x      |
|           | \(\lambda^{Loc}{x}.M\) |
|           | \(\Lambda\alpha.V\) |
|           | \(\Lambda l.V\) |

Semantics

\[
\frac{\lambda^{a}{x}.M \Downarrow a}{\lambda^{a}{x}.M} \quad \text{(Abs)}
\]

\[
\frac{L \Downarrow a \lambda^{b}{x}.N \quad M \Downarrow a W \quad N\{W/x\} \Downarrow b V}{L M \Downarrow a V} \quad \text{(App)}
\]

\[
\frac{\Lambda\alpha.V \Downarrow a \Lambda\alpha.V}{M \Downarrow a \Lambda\alpha.V} \quad \text{(Tabs)}
\]

\[
\frac{M \Downarrow a \Lambda\alpha.V \quad \{B/\alpha\}}{M[B] \Downarrow a V\{B/\alpha\}} \quad \text{(Tapp)}
\]

\[
\frac{M \Downarrow a \Lambda l.V}{M[l] \Downarrow a V\{l/l\}} \quad \text{(Lapp)}
\]

Figure 4: The polymorphic RPC calculus \(\lambda^{\gamma}_{pc}\).

syntactic object \(Loc\) is introduced to be either a location constant or a location variable.

Every \(\lambda\)-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 \(\lambda\)-abstraction \(\lambda^{b}{x}.M\). 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.

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

\[ A, B, C : \text{base} \mid A \xrightarrow{\text{Loc}} A \mid \alpha \mid \forall \alpha. A \mid \forall \alpha. A \]

Typing Rules

\[
\begin{align*}
(T\text{-Var}) & \quad \Gamma(x) = A \\
& \quad \vdash \text{Loc} x : A
\end{align*}
\]

\[
\begin{align*}
(T\text{-Abs}) & \quad \Gamma, x : A \vdash \text{Loc} M : B \\
& \quad \vdash \text{Loc} \lambda x. M : A \xrightarrow{\text{Loc}} B
\end{align*}
\]

\[
\begin{align*}
(T\text{-App}) & \quad \Gamma \vdash \text{Loc} L : A \xrightarrow{\text{Loc}} B \\
& \quad \Gamma \vdash \text{Loc} M : A \\
& \quad \vdash \text{Loc} L M : B
\end{align*}
\]

\[
\begin{align*}
(T\text{-Tabs}) & \quad \Gamma, \alpha \vdash \text{Loc} V : A \\
& \quad \vdash \text{Loc} \Lambda \alpha. V : \forall \alpha. A
\end{align*}
\]

\[
\begin{align*}
(T\text{-App}) & \quad \Gamma \vdash \text{Loc} M : \forall \alpha. A \\
& \quad \vdash \text{Loc} M[B] : A\{B/\alpha\}
\end{align*}
\]

\[
\begin{align*}
(T\text{-Labs}) & \quad \Gamma, l \vdash \text{Loc} V : A \\
& \quad \vdash \text{Loc} \Lambda l. V : \forall l A
\end{align*}
\]

\[
\begin{align*}
(T\text{-Lapp}) & \quad \Gamma \vdash \text{Loc} M : \forall l A \\
& \quad \vdash \text{Loc} M[\text{Loc}'] : A\{\text{Loc}' / l\}
\end{align*}
\]

Figure 5: A type system for the polymorphic RPC calculus

where the definition of \( \text{Loc}\{\text{Loc}' / l\} \) is

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

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

For example, a polymorphic identity function \( \text{id} \) is defined as \( \Lambda l. \lambda x. x \) so that we can use \( \text{id}[c] \) as \( \lambda c x. x \), and \( \text{id}[s] \) as \( \lambda s x. x \). Thus, every polymorphic \( \lambda \)-abstraction can be regarded as a function usable both in the client and in the server, and a choice of a location specific \( \lambda \)-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 \( \text{Loc} \) as their annotation as \( A \xrightarrow{\text{Loc}} B \). Then every \( \lambda \)-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 \( \Gamma \vdash \text{Loc} M : A \) 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 \( \lambda x.M \). Second, typing environments now include location variables as well. A general form of \( \Gamma \) 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. \( \text{dom}(\Gamma) \) 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 \( \text{rng}(\Gamma) \) is \( \{A_1, \cdots, A_m\} \).

Recall that \( (\lambda^c f. \cdots (\lambda^c h. \cdots f \cdots) \cdots) \) is an ill-typed term in the typed RPC calculus, where \( h \) is of type \( A \xrightarrow{\text{Loc}} B \) and \( g \) is of type \( A \xrightarrow{\text{Loc}} B \). 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 $f[c]$ and $f[s]$ 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.

\[
\begin{align*}
flv(a) &= \{\} \\
flv(l) &= \{l\}
\end{align*}
\]

\[
\begin{align*}
flv(base) &= \emptyset \\
flv(\alpha) &= \emptyset \\
flv(A \xrightarrow{Loc} B) &= flv(A) \cup flv(Loc) \cup flv(B) \\
flv(\forall \alpha.A) &= flv(A) \\
flv(\forall l.A) &= flv(A) \setminus \{l\}
\end{align*}
\]

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

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

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 its bound location variable does not appear in the type environment and in the contextual location. (T-Lapp) substitutes $Loc'$ for all occurrences of a location variable $l$ on $\lambda$-abstractions in $M$.

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

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

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, $\bigcup_{A_i \in rng(\Gamma)} flv(A_i) \cup ftv(M) \cup ftv(A) \subseteq dom(\Gamma)$ where $fv(A)$ or $ftv(M)$ are the sets of free type variables occurring in the type and the term respectively. They can be defined straightforwardly. Third, $\bigcup_{A_i \in rng(\Gamma)} flv(A_i) \cup flv(Loc) \cup ftv(M) \cup ftv(A) \subseteq dom(\Gamma)$. 


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 $\alpha$ does not occur as a free type variable in $\Gamma$, and in (T-Labs), a bound location variable $l$ never occurs as a free location variable in $\Gamma$ 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, which is available below. The value substitution lemma (3.2) offers a typing hypothesis for the $\beta$-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 a 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 $\Gamma \vdash Loc\ \lambda x.M : A \xrightarrow{Loc'}\ B$ and $\Gamma \vdash Loc\ V : A$ then $\Gamma \vdash Loc\ \{V/x\} : B$.

**Lemma 3.3** (Type substitution). If $\Gamma \vdash Loc\ \forall \alpha.V : \forall \alpha.A$ 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 $\emptyset \vdash M : A$ and $M \Downarrow V$, then $\emptyset \vdash V : A$.

**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):** $\lambda x.M_0 \Downarrow V$ where $M = \lambda x.M_0$. Therefore, this case holds by the typing judgment hypothesis.

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

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

**Case (App):** $L M_1 \Downarrow V$, (1):$L \Downarrow b$, (2):$M_1 \Downarrow W$, and (3):$M_1\{W/x\} \Downarrow b$ V where $M = L M_1$. By $\emptyset \vdash L M_1 : B$, (4):$\emptyset \vdash L : A_1 \xrightarrow{Loc'}\ B$, and (5):$\emptyset \vdash M_1 : A_1$ where $B = A$.

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

11
From (6) and (7), (8): ∅ ⊢ b \( M_0 \{ W/x \} : B \) is implied by the lemma (Value substitution).

By applying I.H. to (3) and (8), (9): ∅ ⊢ V : B. By the lemma (Value relocation) with (9), ∅ ⊢ b \( M_0 \} : B \). Since \( B = A \), this case is proved.

**Case (Lapp):** \( L[b] \downarrow_\alpha V_0 \{ b/\alpha \} \), (1): \( L \downarrow_\alpha \lambda_\alpha. V_0 \) where \( M = L[b] \) and \( V = V_0 \{ b/\alpha \} \).

∅ ⊢ a \( M_1 \{ B/\alpha \} : A_0 \{ B/\alpha \} \), (2): ∅ ⊢ a \( M_1 : \forall_\alpha. A_0 \) where \( A = A_0 \{ B/\alpha \} \).

By applying I.H. to (1) and (2), (3): ∅ ⊢ a \( \lambda_\alpha. V_0 : \forall_\alpha. A_0 \).

The lemma (Location substitution) with (3) implies (4): ∅ ⊢ a \( V_0 \{ b/\alpha \} : A_0 \{ b/\alpha \} \), which proves this case.

**Case (Tapp):** \( M_1[B] \downarrow_\alpha V_0 \{ B/\alpha \} \), (1): \( M_1 \downarrow_\alpha \lambda_\alpha. V_0 \) where \( M = M_1[B] \) and \( V = V_0 \{ B/\alpha \} \).

∅ ⊢ a \( M_1[B] : A_0 \{ B/\alpha \} \), (2): ∅ ⊢ a \( M_1 : \forall_\alpha. A_0 \) where \( A = A_0 \{ B/\alpha \} \).

By applying I.H. to (1) and (2), (3): ∅ ⊢ a \( \lambda_\alpha. V_0 : \forall_\alpha. A_0 \).

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

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 \( id, \lambda_\alpha. \lambda_\alpha. x. x \), is translated as \( (\lambda_\alpha.x.x, \lambda_\alpha.x.x) \). Also, \( id[c] \) and \( id[s] \) 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 will not be a big problem in practice because the nesting depth of location lambdas is expected to be low. This issue will be discussed later.

For the target of the translation, we extend the typed RPC calculus with pairs \((M, M)\) and projections \( \pi(M) \) 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 \( \lambda_{rpc} \).

We present a monomorphization translation of \( \lambda_{rpc}^p \) into \( \lambda_{rpc} \) as shown in Figure 8. The translation has two parts. The first part is for types. The
Syntax

Location  \( a, b \)  ::=  c  |  s

Term  \( L, M, N \)  ::=  V  |  \( L M \)  |  \( M[A] \)

\[
\begin{align*}
(\text{Term}) & \quad \text{::=} & \quad V & \quad | & \quad L M & \quad | & \quad M[A] \\
& & \text{|} & \quad \pi_i(M)
\end{align*}
\]

Value  \( V, W \)  ::=  x  |  \( \lambda a \ M \)  |  \( \Lambda \alpha.V \)  |  \( (V, V) \)

\[
\begin{align*}
(\text{Value}) & \quad \text{::=} & \quad x & \quad | & \quad \lambda a \ M & \quad | & \quad \Lambda \alpha.V & \quad | & \quad (V, V)
\end{align*}
\]

Semantics

\[
\begin{align*}
\lambda^a x. M  & \quad \Downarrow  \lambda^a x. M & \quad \text{(Abs)} \\
\end{align*}
\]

\[
\begin{align*}
L  & \quad \Downarrow  \lambda^a x. N & \quad M  \quad \Downarrow  \lambda^a W & \quad N\{W/x\}  \quad \Downarrow  V & \quad \text{(App)} \\
L \quad \Downarrow  \lambda^a V & \quad \text{(Abs)} \\
\Lambda \alpha.V  & \quad \Downarrow  \lambda^a \Lambda \alpha.V & \quad \text{(Tabs)} \\
M[B]  & \quad \Downarrow  \lambda^a V\{B/\alpha\} & \quad \text{(Tapp)} \\
L \quad \Downarrow  \lambda^a V & \quad \text{(Abs)} \\
M  \quad \Downarrow  \lambda^a W & \quad \text{(Abs)} \\
(L, M)  & \quad \Downarrow  \lambda^a (V, W) & \quad \text{(Pair)} \\
\end{align*}
\]

\[
\begin{align*}
\pi_i(M)  & \quad \Downarrow  \lambda^a V_i & \quad \text{(Proj-i)} \\
\end{align*}
\]

Figure 6: The typed RPC calculus extended with pairs

type translation of base is base. We translate a monomorphic function type 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 \( c \) or \( s \) 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 \([\forall l.A]\) = ([\([A\{c/l\}],[A\{s/l\}]\)]), 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, \( \lambda \)-abstraction, \( \lambda \)-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: \([\Lambda l.V]\) = ([\([V\{c/l\}],[V\{s/l\}]\)]). 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 \( c \) or \( s \) for every occurrence of the location variable \( l \) in the body \( V \). The translation of location application with \( c \) or \( s \) 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 \( a = c \) and \( i = 2 \) if \( a = s \).

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 location variables.
Types

\[ \text{Type } A, B, C \ ::= \text{base} \mid A \xrightarrow{\alpha} A \mid \alpha \mid \forall \alpha.A \mid A \times A \]

Typing Rules

\[ \Gamma(x) = A \quad \text{(T-Var)} \]
\[ \Gamma \vdash_a x : A \]

\[ \Gamma \vdash_a \lambda x.M : A \quad \text{(T-Abs)} \]
\[ \Gamma \vdash_a \alpha \quad \text{and} \quad \Gamma \vdash_a \forall \alpha.A \]
\[ \Gamma \vdash_a \alpha \quad \text{and} \quad \Gamma \vdash_a \alpha \quad \text{(T-Tabs)} \]
\[ \Gamma \vdash_a M : A \quad \text{and} \quad \Gamma \vdash_a M : B \quad \text{(T-App)} \]
\[ \Gamma \vdash_a M \quad \text{and} \quad \Gamma \vdash_a \forall \alpha.A \quad \text{(T-Tapp)} \]
\[ \Gamma \vdash_a M : \forall \alpha.A \quad \text{and} \quad \Gamma \vdash_a \forall \alpha.A \quad \text{(T-Tapp)} \]
\[ \Gamma \vdash_a M : A \quad \text{and} \quad \Gamma \vdash_a \forall \alpha.A \quad \text{(T-Tapp)} \]

Translation: types

\[ [\text{base}] = \text{base} \quad [A \xrightarrow{\alpha} B] = [A] \xrightarrow{\alpha} [B] \quad [\forall \alpha.A] = [A[c/l]] \times [A[s/l]] \]
\[ [\alpha] = \alpha \quad [\forall \alpha.A] = \forall \alpha.[A] \]

Translation: terms

\[ [x] = x \quad [\lambda \alpha.x.M] = \lambda \alpha.[M] \quad [\forall \alpha.A] = [\forall \alpha.[M]] \]
\[ [L M] = [L] [M] \quad [M[e]] = \pi_1([M]) \quad [M[s]] = \pi_2([M]) \]
\[ [\forall \alpha.A] = \forall \alpha.[V] \quad [M[B]] = [M][B] \]

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

3.2.1. Examples

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

\[ [\forall \lambda.l.x.x] = (([\lambda.l.x.x][c/l]), ([\lambda.l.x.x][s/l])) \]
\[ = ([\lambda.l.x.x][c/l], [\lambda.l.x.x][s/l]) \]
\[ = ([\lambda.l.x.x][c/l], [\lambda.l.x.x][s/l]) \]

Next are more complex examples over map functions. Suppose \( M_X \) is a body of a map function parameterized by location arguments \( X \) as: case \( x \) of \{ [] ⇒ [] ; (y : ys) ⇒ f y :: map[X] f ys \} with some syntactic extension with case expression, list patterns, and list constructors. Also assume that \( [A] \) is assumed to be 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:

\[
\text{letrec \ } map = \Lambda l. \lambda l. f. \lambda l. M \ \text{in} \ \cdots
\]

The translation of this map function proceeds as

\[
[map] = ([(\lambda f. \lambda x. M)\{c/l\}], [(\lambda f. \lambda x. M)\{s/l\}])
\]

\[
= (\lambda f. [\lambda x. M_0]. \lambda l. f. [\lambda x. M_s])
\]

\[
= (\lambda f. \lambda l. [M_c]. \lambda s. f. \lambda x. [M_s]).
\]

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, \(map[c]\) in \(M_c\) is translated as \(\pi_1([map])\), and, in the second element, \(map[s]\) in \(M_s\) is translated as \(\pi_2([map])\). Here, instead of repeating \([map]\) from the beginning, the translation stops here to construct a recursive function, respectively. 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 \(\forall l_1. \forall l_2. \forall l_3. (A \xrightarrow{l_1} B) \xrightarrow{l_2} ([A] \xrightarrow{l_3} [B])\) can be defined as:

\[
\text{letrec \ } map = \Lambda l_1. \Lambda l_2. \Lambda l_3. map_0 \ \text{in} \ \cdots
\]

where \(map_0\) is \(\lambda l_1. \lambda l_2. \lambda l_3. M_{l_1 l_2 l_3}\) and multi location applications \(M[Loc_1 \cdots Loc_k]\) is defined as \(M[Loc_1] \cdots [Loc_k]\). The translation \([map]\) becomes

\[
\begin{pmatrix}
\text{map}_0(c/l_1, c/l_2, c/l_3), & \text{map}_0(c/l_1, c/l_2, s/l_3), & \text{map}_0(c/l_1, s/l_2, c/l_3), & \text{map}_0(c/l_1, s/l_2, s/l_3), \\
\text{map}_0(c/l_1, s/l_2, c/l_3), & \text{map}_0(c/l_1, s/l_2, s/l_3), & \text{map}_0(s/l_1, c/l_2, c/l_3), & \text{map}_0(s/l_1, c/l_2, s/l_3), \\
\text{map}_0(s/l_1, s/l_2, c/l_3), & \text{map}_0(s/l_1, s/l_2, s/l_3)
\end{pmatrix}
\]

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 \cite{Radanne, 2017} \cite{Radanne and Vouillon, 2018} compiler also uses code duplication to support a kind of location polymorphism, what they call \textit{shared sections} in the expression-level and \textit{mixed modules} in the module-level. In the related work, we will discuss these features in detail.
Second, we could take an 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.

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 $\lambda_{rpc}$. 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:

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

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 $V\{c/l\}$ and $V\{s/l\}$, 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 \( c \) or \( s \) 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 monomorphism). \([A]([B]/\alpha) = [A[\alpha/B]]\).

**Lemma 3.6** (Location polymorphism). Suppose \( \Gamma = \{l_1, \cdots, l_n\} \cup \Gamma_0 \) such that \( \Gamma_0 \) has no location variables. If \( \Gamma \vdash_{\text{Loc}} M : A \) then \( (\Gamma_0 \vdash_{\text{Loc}} M : A\{\alpha_1/l_1, \cdots, \alpha_n/l_n\} \) for any \( \alpha_1, \cdots, \alpha_n \) with the same height.

**Lemma 3.7.** Given \( \text{flv}(\Gamma) \cup \text{flv}(M) \cup \text{flv}(A) = \emptyset \), if \( \Gamma \vdash_{\alpha} M : A \) then \( [[\Gamma]] \vdash_{\alpha} [M] : [A] \).

**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 \( \text{FLV}(\Gamma) = \emptyset \) and \( \text{FLV}(A) = \emptyset \). By the typing derivation, \( \Gamma(x) = A \). This implies \((1) : [\Gamma](x) = [A] \). By (T-Var) with (1), \([\Gamma][x : A] \), which proves this case since \([x] = x \).

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

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

By (T-Abs) with (3), (4): \( \Gamma \vdash_{\alpha} \lambda_{\text{Loc}} x. [M_0] : [A_0] \xrightarrow{\text{Loc}} [B_0] \). By the definition of the translation, (4) implies \([\Gamma] \vdash_{\alpha} [\lambda_{\text{Loc}} x. M_0] : [A_0] \xrightarrow{\text{Loc}} [B_0] \) since \( \text{Loc} = b \).

(T-App): \( \Gamma \vdash_{\alpha} L M_0 : A \), (1): \( \Gamma \vdash_{\alpha} L : B \xrightarrow{\text{Loc}} A \), and (2): \( \Gamma \vdash_{\alpha} M_0 : B \) where \( M = L M_0 \).

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

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

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

By applying I.H. to (2) and (4), \((6) : [\Gamma] \vdash_{\alpha} [M_0] : [B] \).

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

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

(2): \( \text{flv}(\Gamma, \alpha) \cup \text{flv}(V_0) \cup \text{flv}(A_0) = \emptyset \).

By applying I.H. to (2), \( [\Gamma, \alpha] \vdash_{\alpha} [V_0] : [A_0] \), which is \((3) : [\Gamma], \alpha \vdash_{\alpha} [V_0] : [A_0] \) by the def. of the translation.
By (T-Tabs) with (3), \([\Gamma] \vdash_a \text{Abs} \subseteq \exists \alpha.\beta_0 : \forall \alpha.\beta_0\]. This is the same as \([\Gamma] \vdash_a [\alpha \cdot \beta_0] : [\forall \alpha.\beta_0] \) by the def. of the translation, which proves this case.

**(T-Tapp)**: \(\Gamma \vdash_a M_1 \cdot [B] : A_0 \cdot [B]\) and \((\forall \Gamma \vdash_a M_1 : \forall \alpha.\beta_0\) where \(M = M_1 \cdot [B] \) and \(A = A_0 \cdot [B]\).

(2):\(\text{flv}(\Gamma) \cup \text{flv}(M_1) \cup \text{flv}(\forall \alpha.\beta_0) = \emptyset\).

By I.H. with (2), \([\Gamma] \vdash_a [M_1] : [\forall \alpha.\beta_0]\), which is (3):\([\Gamma] \vdash_a [M_1] : [\forall \alpha.\beta_0] \) by the def. of the translation.

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

By the lemma \([3,5]\) with (4), \(\forall \Gamma \vdash_a [M_1] [[B]] : [A_0 \cdot [B / \alpha]]\), which proves this case by the def. of the translation.

**(T-Labs)**: \(\forall \alpha \cdot \beta_0 \subseteq \forall \alpha.\beta_0\) and \((\forall \Gamma \vdash_a \forall \alpha.\beta_0 : \forall \alpha.\beta_0\) where \(M = \forall \alpha.\beta_0\) and \(A = \forall \alpha.\beta_0\).

By applying the lemma \([3,6]\) to (1), \(\forall \alpha \cdot \beta_0 \subseteq \forall \alpha.\beta_0\) for all \(a\). Also, (3):\(\text{flv}(\Gamma) \cup \text{flv}(\forall \alpha.\beta_0) = \emptyset\) since \(l\) is the only free location variable.

By applying I.H. to (2) and (3), \(\forall \alpha \cdot \beta_0 \subseteq \forall \alpha.\beta_0\) for all \(a\). Also, \(\forall \Gamma \vdash_a [V_0(a/l)] : [A_0(a/l)]\) for \(a = c, s\).

By (T-Pair) in the type system for \(\lambda_{\rho_c}\) with two premises, one with (4) and \(a = c\) and the other with (4) and \(a = s\), \([\Gamma] \vdash_a ([V_0(c/l)] :: [V_0(s/l)]) : [A_0(c/l)] \times [A_0(s/l)]\).

By the definition of the translation, \([\Gamma] \vdash_a [\forall \alpha.\beta_0] : [\forall \alpha.\beta_0]\), which proves this case.

**(T-Lapp)**: \(\forall \alpha \cdot \beta_0 \subseteq \forall \alpha.\beta_0\) and \((\forall \Gamma \vdash_a \forall \alpha.\beta_0 : \forall \alpha.\beta_0\) where \(M = L[\beta_0/l] \) and \(A = A_0 \cdot [L[\beta_0/l]]\).

By (T-Pair) in the type system for \(\lambda_{\rho_c}\) with two premises, one with (4) and \(a = c\) and the other with (4) and \(a = s\), \([\Gamma] \vdash_a [L] :: [\forall \alpha.\beta_0]\), which is (3): \([\Gamma] \vdash_a [L] :: [A_0(c/l)] \times [A_0(s/l)]\) by the definition of the translation.

By (T-Proj-i) with (3), \(\forall \pi_i(L) :: [A_0 \cdot [L[\beta_0/l]]\) where \(i = 1\) if \(L[\beta_0/l] = c\), and \(i = 2\) if \(L[\beta_0/l] = s\), which is (4):\([\Gamma] \vdash_a [L[\beta_0/l]] :: [A_0 \cdot [L[\beta_0/l]]\) 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 \(\text{flv}(\emptyset) \cup \text{flv}(M) \cup \text{flv}(A) = \emptyset\).

### 3.2.3. Semantic correctness

This time we formulate a stronger property, the semantic correctness of the monomorphization translation, as Theorem \([3,5]\) for a well-typed closed term, the evaluation of the term in \(\lambda_{\rho_c}\) is preserved in \(\lambda_{\rho_c}\) 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 $b = c$ and the other with $b = s$, both of which can be proved with no difficulty.

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

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

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

**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 $V_i$s 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 $V_0 = \lambda^{loc}.x.M_0$.

$[M] = [V] = [\Lambda l.\lambda^{loc}.x.M_0] = ([\lambda^{loc}.x.M_0]\{c/l\}], ([\lambda^{loc}.x.M_0]\{s/l\})$ which is translated into $(\lambda^{loc}(c/l)x.[M_0\{c/l\}], \lambda^{loc}(s/l)x.[M_0\{s/l\})$. 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, \( \text{Loc}\{a/l\} \) is \( a \).

By (Abs), (1): \( \lambda_{\text{Loc}}(c/l).M_0\{c/l\} \Downarrow_a \lambda_{\text{Loc}}(c/l).M_0\{c/l\} \).

By (Abs), (2): \( \lambda_{\text{Loc}}(s/l).M_0\{s/l\} \Downarrow_a \lambda_{\text{Loc}}(s/l).M_0\{s/l\} \).

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

Also when \( n = 0 \), consider the other base case with \( V_0 = \lambda_{\alpha}W \).

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

which is translated into \( (\lambda_{\alpha}.W)\{c/l\}, \lambda_{\alpha}.[W\{s/l\}] \).

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

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

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, \( n > 0 \).

\[
[M] = [V] = [\Lambda.l.\Lambda_{\alpha}.V_1] = ([[(\Lambda.l.\Lambda_{\alpha}.V_1)\{c/l\}]], [(\Lambda.l.\Lambda_{\alpha}.V_1)\{s/l\}])
\]

which is \( ([M_1.\{V_1\{c/l\}])], [(\Lambda.l.\Lambda_{\alpha}.V_1)\{s/l\}]) \) since \( l \) must not be equal to \( l_1 \). Then, by I.H. with \( n - 1 \) of the second induction,

\[
(3) : [M_1.\{V_1\{c/l\}])], [(\Lambda.l.\Lambda_{\alpha}.V_1)\{s/l\}]) \Downarrow_a [\Lambda.l.\Lambda_{\alpha}.V_1]) \quad (4) : [M_1.\{V_1\{s/l\}])], [(\Lambda.l.\Lambda_{\alpha}.V_1)\{s/l\}]) \Downarrow_a [\Lambda.l.\Lambda_{\alpha}.V_1]
\]

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

\textbf{(App):} \( L_1M_1 \Downarrow_a V, \quad (1):L_1 \Downarrow_a \lambda^b.x.M_0, \quad (2):M_1 \Downarrow_a W, \quad \) and \( (3):M_0\{W/x\} \Downarrow_b V \)

where \( M = L_1M_1 \). By (T-App), (4): \( \emptyset \vdash_a L_1 : B \Downarrow_a A \) 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): \( \emptyset \vdash_a \lambda^b.x.M_0 : B \Downarrow_a A \).

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 \( [L_1M_1] = [L_1][M_1] \).

\textbf{(Tapp):} \( M_1B \Downarrow_a V_0\{B/\alpha\} \), 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):\( L \Downarrow_a M.V_0 \) where \( M = L[b] \) and \( V = V_0{b/l} \). By (T-Lapp), \( \emptyset \vdash L[b] : A_0{b/l} \) and (2):\( \emptyset \vdash a : \forall l.A_0 \) where \( A = A_0{b/l} \).

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

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

\[ \square \]

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:

```plaintext
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

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

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

---

\(^4\)In fact, it is *sham*, a variant of the box modal construct.
instantiated with some current world. In the example, \( w \) becomes instantiated with \textit{server} in the server where \textit{map} is used. The polymorphic RPC calculus expresses location polymorphism by parametric polymorphism. In the example, the client would select a server version of \textit{map} by \textit{map}[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 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 polymorphic 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.

In Eliom (Radanne, 2017; Radanne and Vouillon, 2018), there are a few features to discuss for comparison. First, it supports a macro feature called \textit{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 \textit{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 \textit{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, \( \lambda^c(f : A \rightarrow B), \cdots \) \((\lambda^c x. \cdots)\) is ill-typed but the term can be slightly changed to be well-typed by inserting an \( \eta \)-conversion coercion over the argument automatically as: \( \lambda^c(f : A \rightarrow B), \cdots \) \((\lambda^c y.(\lambda^c x. \cdots) y)\).

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 for Scala (Reynders et al., 2014); and ScalaLoci (Weisenburger et al., 2018).
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 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 together with proofs on 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.

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 asyn-
chronous session types: Session types without tiers. Proc. ACM Pro-
gram. 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 Sym-
posium 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, Ger-
many. 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 integra-
tion, 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 pro-
gramming 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
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 $\Gamma \vdash_{\text{Loc}} V : A$ then $\Gamma \vdash_{\text{Loc}'} V : A$.

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 $\text{Loc}$ this time, $\Gamma \vdash_{\text{Loc}} x : A$. 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):$\Gamma, \alpha \vdash_{\text{Loc}} V_0 : A_0$.

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

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

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

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

Proof. By (T-Abs) with the first part of the condition, $\Gamma, x : A \vdash_{\text{Loc}'} M : B$.

Since $x$ is a bound variable, $x$ cannot appear as a free variable in $V$. By the lemma (Value relocation), we have $\Gamma \vdash_{\text{Loc}'} V : A$ from the second part.

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

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 $\Gamma, x : A \vdash_{\text{Loc}'} L N : B$.

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

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

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

By (T-App) with (7) and (8), $\Gamma \vdash_{\text{Loc}'} (L[V/x]) (N[V/x]) : B$ where $(L[V/x]) (N[V/x]) = (L N)[V/x]$. 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 $\Gamma \vdash_{\text{Loc}} \Lambda \alpha.V : \forall \alpha.A$ then $\Gamma \vdash_{\text{Loc}} V\{B/\alpha\} : A\{B/\alpha\}$.

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

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

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, $\alpha$ 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):$\Gamma, \alpha \vdash_{\text{Loc}} L[C] : A_0\{C/\beta\}$. By (T-Tapp) with (4), we have (5):$\Gamma, \alpha \vdash_{\text{Loc}} L : \forall \beta.A_0$. By I.H. with (5), (6):$\Gamma \vdash_{\text{Loc}} L\{B/\alpha\} : (\forall \beta.A_0\{B/\alpha\})$. 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):$\Gamma \vdash_{\text{Loc}} L\{B/\alpha\}C\{B/\alpha\} : (A_0\{B/\alpha\})C\{B/\alpha\}/\beta$. This proves the inductive case by $\Gamma \vdash_{\text{Loc}} (L[C])\{B/\alpha\} : (A_0\{C/\beta\})\{B/\alpha\}$. The other inductive cases use (T-Abs), (T-App), (T-Tabs), (T-Lapp), and (T-Lapps), which are provable similarly.

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

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

We prove a generalized lemma as: if (1):$\Gamma, l \vdash_{\text{Loc}} M : A$ and (2):$l \notin \text{ftv}(\Gamma)$ then (3):$\Gamma \vdash_{\text{Loc}} M\{Loc'/l\} : A\{Loc'/l\}$. 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):$\Gamma, l \vdash_{\text{Loc}} L[Loc_0] : B\{Loc_0/l_0\}$.

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

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

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

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

The other inductive cases use (T-Abs), (T-App), (T-Tabs), (T-Tapp), and (T-Lapps), 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 monomorphism). $[A][[B]/\alpha] = [A\{B/\alpha\}]$. 

28
Proof. We prove this lemma by the structural induction on \( A \). We have two base cases. Let us first consider when \( A = \text{base} \). In the left-hand side of the equation, \([\text{base}]\{[B]/\alpha\} = \text{base}\{[B]/\alpha\} = \text{base}\). In the right-hand side, \([\text{base}\{B/\alpha\}] = \text{base}\).

For the other base case that \( A \) is a type variable, say, \( \beta \), we do a case analysis on if \( \beta \) is the same as \( \alpha \) 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** \( A_1 \overset{\alpha}{\to} A_2 \):

\[
\begin{align*}
[A_1 \overset{\alpha}{\to} A_2]\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= ([A_1] \overset{\alpha}{\to} [A_2])\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= [A_1]\{[B]/\alpha\} \overset{\alpha}{\to} [A_2]\{[B]/\alpha\} & \quad \text{by I.H.} \\
= [A_1\{B/\alpha\}] \overset{\alpha}{\to} [A_2\{B/\alpha\}] & \quad \text{by def. of } [-] \\
= ([A_1 \overset{\alpha}{\to} A_2]\{B/\alpha\}) & \quad \text{by def. of } [-]
\end{align*}
\]

**Case** \( A \) **is** \( \forall \beta. A_0 \):

i) \( \beta = \alpha \)

\[
\begin{align*}
[\forall \beta. A_0]\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= (\forall \beta. [A_0])\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= \forall \beta. [A_0] & \quad \text{by def. of } [-] \\
= \forall \beta. A_0 & \quad \text{by def. of } [-] \\
= \{(\forall \beta. A_0)\{B/\alpha\}\}
\end{align*}
\]

ii) \( \beta \neq \alpha \)

\[
\begin{align*}
[\forall \beta. A_0]\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= (\forall \beta. [A_0])\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= \forall \beta. ([A_0])\{[B]/\alpha\} & \quad \text{by I.H.} \\
= \forall \beta. ([A_0\{B/\alpha\}]) & \quad \text{by I.H.} \\
= \forall \beta. (A_0\{B/\alpha\}) & \quad \text{by I.H.} \\
= \{(\forall \beta. A_0)\{B/\alpha\}\}
\end{align*}
\]

**Case** \( A \) **is** \( \forall l. A_0 \):

\[
\begin{align*}
[\forall l. A_0]\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= ([A_0\{c/l\}] \times [A_0\{s/l\}])\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= ([A_0\{c/l\}] \times [A_0\{s/l\}])\{[B]/\alpha\} & \quad \text{by def. of } [-] \\
= [A_0\{c/l\}]\{[B]/\alpha\} \times [A_0\{s/l\}])\{[B]/\alpha\}] & \quad \text{by I.H.} \\
= ([A_0\{c/l\}]\{B/\alpha\} \times [A_0\{s/l\}])\{B/\alpha\}] & \quad \text{by I.H.} \\
= ([A_0\{B/\alpha\}]\{c/l\}) \times ([A_0\{B/\alpha\}]\{s/l\} & \quad \text{by def. of } [-] \\
= [\forall l. A_0\{B/\alpha\}] & \quad \text{by def. of } [-] \\
= \{(\forall l. A_0)\{B/\alpha\}\}
\end{align*}
\]

\[\square\]

**Lemma 3.6** (Location polymorphism). Suppose \( \Gamma = \{l_1, \cdots, l_n\} \cup \Gamma_0 \) such that \( \Gamma_0 \) has no location variables. If \( \Gamma \vdash_{\text{Loc}} M : A \) then \( \Gamma_0 \vdash_{\text{Loc}} M : A \).
Lemma 3.8

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

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

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) \(y = x\)

\[ \lambda^a y. L[[W]/x]] = (\lambda^a y. L)[W]/x\\]
\[ = \lambda^a y. L\\]
\[ = [[\lambda^a y. L]/x]] \]

ii) \(y \neq x\)

\[ \lambda^a y. L[[W]/x]] = (\lambda^a y. L)[W]/x\\]
\[ = \lambda^a y. (L[[W]/x]] \]
\[ = [[\lambda^a y. L]/x]] \]

Case \(M\) is \(LN\\):

\[ LN[[W]/x]] = (LN)[W]/x\\]
\[ = (L)[N]/[W]/x\\]
\[ = (L)[W/x])([N][W/x]) \]
\[ = (L[W/x])(N[W/x])\\]
\[ = [[LN]/W/x]] \]

Case \(M\) is \(\Lambda^a V\\):

\[ \Lambda^a V[[W]/x]] = (\Lambda^a V)[W]/x\\]
\[ = (\Lambda^a V)[c/l], [V{s/l}]/[W]/x\\]
\[ = ([V][c/l])[W]/x], [V{s/l}]/[W]/x] \]
\[ = ([[V][c/l]W/x], [[V][s/l]W/x]] \]
\[ = (\Lambda^a V)[c/l], [[V]/W/x][s/l]\\]
\[ = (\Lambda^a V)[W/x]] \]

Case \(M\) is \(\Lambda a. V\\):

\(30\\)
\[ [\Lambda \alpha. V]([W]/x) \quad \text{by def. of } [-] \]
\[ = (\Lambda \alpha. [V])([W]/x) \]
\[ = \Lambda \alpha. (V([W]/x)) \quad \text{by I.H.} \]
\[ = \Lambda \alpha. (V(W/x)) \quad \text{by def. of } [-] \]
\[ = ([\Lambda \alpha. V](W/x)) \]

Case \( M \) is \( L[a] \):
\[ [L[a]]([W]/x) \quad \text{by def. of } [-] \]
\[ = (\pi_i[L])([W]/x) \quad i = 1 \text{ if } a = c, \ i = 2 \text{ if } a = s \]
\[ = \pi_i([L(W/x)]) \quad \text{by I.H.} \]
\[ = [L(W/x)[a]] \quad \text{by def. of } [-] \]
\[ = [(L[a])(W/x)] \]

Case \( M \) is \( L[B] \):
\[ [L[B]]([W]/x) \quad \text{by def. of } [-] \]
\[ = [L][B]([W]/x) \]
\[ = ([L][W/x])[B] \quad \text{by I.H.} \]
\[ = [L(W/x)][B] \quad \text{by def. of } [-] \]
\[ = [(L[B])(W/x)] \]

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\{B/\alpha\}\]. \)

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

Case \( M \) is \( \lambda^a.x.L \):
\[ [\lambda^a.x.L][[B]/\alpha] \quad \text{by def. of } [-] \]
\[ = (\lambda^a.x.[L][B]/\alpha) \]
\[ = \lambda^a.x.(L([B]/\alpha)) \quad \text{by I.H.} \]
\[ = \lambda^a.x.[L(B/\alpha)] \]
\[ = [[\lambda^a.x.(L(B/\alpha))]] \quad \text{by def. of } [-] \]
\[ = [\lambda^a.x(L(B/\alpha))] \]

Case \( M \) is \( \Lambda l.V \):
\[\begin{align*}
[\Lambda.M.V] & \{[B]/\alpha\} & \text{by def. of \([-\]} \\
= & \ (\{V\{c/l\}\}_i, \{V\{s/l\}\}_i),\{[B]/\alpha\} & \\
= & \ (\{V\{c/l\}\}_i,\{[B]/\alpha\}, \{V\{s/l\}\}_i),\{[B]/\alpha\}) & \text{by applying I.H. twice} \\
= & \ (\{L\{V\{B/\alpha\}\}_i\{c/l\}\}, \{V\{B/\alpha\}\}_i\{s/l\}) & \text{by def. of \([-\]} \\
= & \ [[[\Lambda.M.(V\{B/\alpha\})] & \\
= & \ [[[\Lambda.M.V\{B/\alpha\}] & \\

\text{Case } M \text{ is } LN:\n
\begin{align*}
[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 \([-\]} \\
= & \ [[LN]\{B/\alpha\}] & \\

\text{Case } M \text{ is } L[a]:
\begin{align*}
[L[a]] & \{[B]/\alpha\} & \text{by def. of \([-\]} \ (i = 1 \text{ if } a = c, \ i = 2 \text{ if } a = s) \\
= & \ (\pi_i[L])\{[B]/\alpha\} & \\
= & \ \pi_i[L\{[B/\alpha]\}] & \text{by I.H.} \\
= & \ [\{L\{B/\alpha\}\}]\{a\} & \text{by def. of \([-\]} \\
= & \ [[L[a]\{B/\alpha\}] & \\

\text{Case } M \text{ is } \Lambda\beta.V:
\begin{enumerate}
i) \ \beta = \alpha
\begin{align*}
[\Lambda\alpha.V] & \{[B]/\alpha\} & \text{by def. of \([-\]} \\
= & \ (\Lambda\alpha.V)\{[[B]/\alpha\} & \\
= & \ \Lambda\alpha.V & \text{by def. of \([-\]} \\
= & \ [[\Lambda\alpha.V]\{B/\alpha\}] & \\

\text{i) } \ \beta \neq \alpha
\begin{align*}
[\Lambda\beta.V] & \{[B]/\alpha\} & \text{by def. of \([-\]} \\
= & \ (\Lambda\beta.V)\{[[B]/\alpha\} & \\
= & \ \Lambda\beta.V & \text{by I.H.} \\
= & \ [[\Lambda\beta.V]\{B/\alpha\}] & \\
= & \ [[\Lambda\beta.V]\{B/\alpha\}] & \\

\text{Case } M \text{ is } L[A]:
\end{enumerate}
\end{align*}
\end{enumerate}
\[
\begin{align*}
&= [L[A]][[B]/\alpha] & \text{by def. of } [-] \\
&= ([L][A])[B/\alpha] & \\
&= ([L][B/\alpha])[A][B/\alpha] & \text{by I.H. and Lemma 3.5} \\
&= [L(B/\alpha)][A(B/\alpha)] & \text{by def. of } [-] \\
&= [L(B/\alpha); A(B/\alpha)] \\
&= [(L[A])B/\alpha] \\
&= (L[A])B/\alpha \\
&\square
\end{align*}
\]