A Typed Slicing Compilation of the Polymorphic RPC calculus

Kwanghoon Choi
Chonnam National University
Gwangju, Republic of Korea
kwanghoon.choi@jnu.ac.kr

James Cheney
The University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Sam Lindley
The University of Edinburgh
Edinburgh, United Kingdom
Sam.Lindley@ed.ac.uk

Bob Reynders
Chonnam National University
Gwangju, Republic of Korea
tzbob@gmail.com

ABSTRACT

The polymorphic RPC calculus allows programmers to write succinct multitier programs using polymorphic location constructs. However, until now it lacked an implementation. We develop an experimental programming language based on the polymorphic RPC calculus. We introduce a polymorphic Client-Server (CS) calculus with the client and server parts separated. In contrast to existing untyped CS calculi, our calculus is not only able to resolve polymorphic locations statically, but it is also able to do so dynamically. We design a type-based slicing compilation of the polymorphic RPC calculus into this CS calculus, proving type and semantic correctness. We propose a method to erase types unnecessary for execution but retaining locations at runtime by translating the polymorphic CS calculus into an untyped CS calculus, proving semantic correctness.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Distributed programming languages**; **Client-server architectures**.

KEYWORDS

multi-tier programming language, polymorphism, rpc calculus, client-server calculus, slicing compilation

ACM Reference Format:
Kwanghoon Choi, James Cheney, Sam Lindley, and Bob Reynders. 2021. A Typed Slicing Compilation of the Polymorphic RPC calculus. In *Proceedings of the 23rd Symposium on Principles and Practice of Declarative Programming, PPDP 2021, Tallinn, Estonia, September 6–8, 2021.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3479394.3479406

1 INTRODUCTION

Multi-tier programming languages address the complexity of developing distributed systems by providing abstractions for communication between peers. For instance, a web application is a basic distributed system consisting of a client, which provides access to a user interface, and a server, which provides access to a persistent database, where the client and server are connected by the HTTP network protocol. Typically, the client and server code must be developed as two separate programs and run on two separate machines, adding to the programmer's burden. The two programs must be tested together, which is more complex than testing one program on a single machine. As the web application evolves, suitable invariants between client and server programs must be carefully maintained. Worse, when tasks cross the boundary of the client and server, the programmer must split the work across the two programs, often baking in implementation decisions that are hard to understand, revisit or maintain.

Multi-tier programming solves this problem by allowing programmers to write client and server expressions together in a single programming language, and by automatically slicing the unified program into client and server programs that are connected together with networking libraries automatically.

An important feature of multi-tier programming languages is the ability to specify locations where code should run. RPC calculi [4, 5, 8] offer a promising, yet lightweight, semantic foundation for multi-tier programming: Firstly, programmers have only to add location annotations, for example, **c** for client and **s** for server, to lambda abstractions to write remote procedures such as $\lambda^{\mathbf{c}}x.M$ and $\lambda^{\mathbf{s}}x.M$. Secondly, remote procedure calls are as simple as local calls, reusing the standard function application syntax. Thirdly, RPC calculi allow unrestricted composition of differently located procedures. These features are not provided by existing multi-tier programming languages such as ML5 [16, 17], Eliom [21, 23], Hop [26–28], Ur/Web [3], ScalaLoci [29, 30], and Gavial [25].

However, the simplicity of RPC calculi gives rise to a difficult choice between convenience and efficiency: it can be hard to determine statically whether a given call site is local or remote. Links [7], a practical multi-tier web programming language, is based on the untyped RPC calculus [8], which provides no static location information and thus depends entirely on runtime location-checking. As function calls are pervasive in functional languages, even a small overhead may be costly. For instance, local computations that could run efficiently on the server may see a significant slow-down as a result of having to dynamically check whether a client call is required, even when the check always determines that a client call is unnecessary. Given compile-time location information at each call site, such overheads can be avoided.

The simply-typed RPC calculus [4] is designed to offer complete location information statically through types in order to determine

PPDP '21, September 6–8, 2021, Tallinn, Estonia

ACM ISBN 978-1-4503-8689-0…$15.00
https://doi.org/10.1145/3479394.3479406

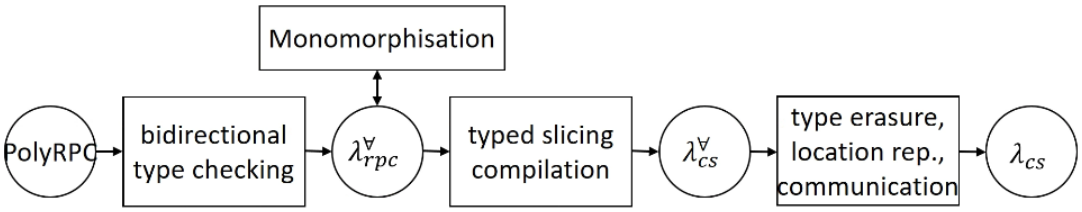

Figure 1: A PolyRPC Compiler

all remote procedure calls at compile-time. Such typed location information is not only useful for avoiding the overhead of runtime location checking but also guides the design of simple slicing compilation methods for both stateless and stateful client-server calculi. We anticipate further applications of static location information to multi-tier programming in the future. Despite this prospect, the simply-typed RPC calculus only allows programmers to statically specify fixed locations; it does not support polymorphic locations, which are useful for writing succinct programs. For instance, Eliom [21–24] provides a macro feature to make it possible to write code for the client and for the server at the same time; polymorphic locations offer similar functionality.

In previous work introducing the polymorphic RPC calculus, $\lambda^{\forall}_{rpc}$, [5], it was proposed to implement polymorphic locations by a so called *monomorphisation* translation, which translates polymorphically located programs into monomorphic ones at compile time, by specialising each location-polymorphic function and compiling it once for each possible location assignment of concrete locations to polymorphic location variables. On top of existing monomorphically typed RPC calculi [4], this translation could be used as the basis for an implementation of the polymorphic RPC calculus. Although the previous work did not provide or evaluate an implementation, but one clear concern about monomorphisation is that it can lead to a code explosion problem.

In this paper, we design a *polymorphic* Client-Server (CS) calculus, $\lambda^{\forall}_{cs}$, and a *type-based* slicing compilation of the polymorphic RPC calculus into $\lambda^{\forall}_{cs}$. Prior CS calculi [4, 5, 7] and their slicing compilers are untyped. In (typed or untyped) CS calculi, the client part is clearly separated from the server part, and communication between the two is inserted automatically.

The first highlight of the polymorphic CS calculus is that the type system guarantees that while functions may be passed to arbitrary locations, every function must run at the declared location. Regardless of how slicing compilation is specified, the type system ensures that first-class functions are well-behaved in located contexts.

The second highlight is that our polymorphic CS calculus is designed to support the combination of the static approach relying on monomorphisation and a complementary dynamic approach for handling polymorphic-location programs directly. In the dynamic approach, locations are passed and examined at runtime, thus avoiding the code explosion problem.

The idea of dynamically passing locations is reminiscent of intentional polymorphism [14] and type-erasure semantics [9]. Accordingly, we propose an efficient implementation strategy for polymorphic CS calculus by erasing all types unnecessary for computation, but retaining those locations required at runtime. We introduce explicit CS communication primitives.

The third highlight is a monadic abstraction for *trampolined style* RPC communication where a single "scheduler" loop, called trampoline, manages all transfers of control by remote procedure call. This allows us to treat the polymorphic CS calculus like a sequential calculus over the client and server.

For a practical aspect, we design an experimental multi-tier programming language for the Web, named *PolyRPC*, based on the polymorphic RPC calculus and implement its compiler based on the polymorphic CS calculus as shown in Figure 1. In the language, the calculi are extended with basic programming features, such as recursion, data types, and references. In our PolyRPC compiler, the front-end is equipped with a simple bidirectional type checker [10]. Monomorphisation is implemented, and it can be enabled or disabled. When monomorphisation is disabled, polymorphic locations are resolved dynamically. The back-end comprises a slicing compiler for the polymorphic CS calculus, followed by type-erasure and location representation stages with the introduction of explicit communication primitives for the CS based Web system.

Using this programming system, we validate the usefulness of polymorphic locations by developing a multi-tier ToDo list program.

The contributions of this paper are summarized as follows:

- For a polymorphic RPC calculus, we introduce a new polymorphic CS calculus, $\lambda^{\forall}_{cs}$, and prove type-soundness.
- We design a typed slicing compilation of $\lambda^{\forall}_{rpc}$ into $\lambda^{\forall}_{cs}$ via a static approach and a dynamic approach, and prove its type correctness and semantic correctness.
- We describe an implementation of $\lambda^{\forall}_{cs}$ by erasing types but retaining locations in terms and by making client-server communication explicit, and we prove semantic correctness.
- We design and implement an experimental multi-tier programming language for the Web, and discuss a case study with a multi-tier ToDoMVC program.

Section 2 presents a case study to help understand the polymorphic RPC calculus in practice. Section 3 gives a formal account of the polymorphic RPC calculus. Section 4 proposes a polymorphic client-server calculus, proves type-soundness of this calculus, describes a typed slicing compilation of RPC calculus into CS calculus, and proves type and semantic correctness of compilation. Section 5 details how to implement the polymorphic CS calculus using type-erasure. Related work is discussed in Section 6 and Section 7 concludes. Proofs are available in the extended version [6].

2 CASE STUDY: A MULTI-TIER TODO LIST

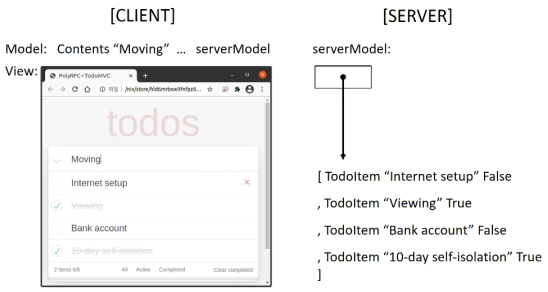

Figure 2: Running the Multi-tier TodoMVC Program

In this section we illustrate polymorphic RPC calculus with an example multi-tier web application. Our example is the 'Hello world' of web programming: a todo list application. The TodoMVC program manages a list of work items, and is structured using the Model-View-Update (MVU [12]) pattern. It is written in *PolyRPC*, an experimental programming language based on the polymorphic RPC calculus in Section 3.

The multi-tier TodoMVC program consists of a web-based UI on the client, and a model for managing the items on the server. The UI allows a user to ask the server to add a new item, mark an item as completed, and delete an item. Figure 2 shows the program running and depicts the configuration of the client and the server.

We present source code for TodoMVC in Figure 3. A longer version, complete with CSS styling, is available online[1]. Following the MVU design pattern, the main value (Line 42) declares a page with the initial model, a view function, and an updating function.

A model of type `Model` (Line 2) is a triple of a text string that the user is typing, a list of visible items of type `List [TodoItem]`, and a reference to a list of all items at the server of type `Ref {server} [List [TodoItem]]` where `Ref {server}` is the location application type of `Ref` to `server` using a notation $\{-\}$ and `List [TodoItem]` is the type application type of `List` to `TodoItem` using another notation $[-]$ in PolyRPC.

The `view` function (Line 18) takes a model and returns an HTML value. The `client` annotation on the function type ensures that the function is run on the client.

A user interacts with the constructed HTML of type `Html [Msg]` through event handler actions that generate messages of type `Msg`. Each message is handled by the `update` function (Line 24) which runs on the client and updates the existing model using references, according to the message.

A locative and polymorphic reference type `Ref {Loc} [A]` is an abstract data type parameterized by locations `Loc`, as well as types `A`, with three interface functions

- `ref : {l}. [a]. a -l-> Ref {l} [a]`
- `(!) : {l}. [a]. Ref {l} [a] -l-> a`
- `(:=) : {l}. [a]. Ref {l} [a] -l-> a -l-> Unit .`

where `{l}.A` is a location abstraction type over a location variable `l` and `[a].A` is a type abstraction type over a type variable `a`. The server model is represented as a reference to a work item list stored on the server is `Ref {server} [List [TodoItem]]`. It is initialised to an empty list (Line 40). Only `ref {server}`, `! {server}`, and `:= {server}` can create, read, and modify server references. We write `M {Loc}` for the location application of M to Loc.

A key property is that every reference of type `Ref {Loc} [A]` is dereferenced only at the right location `Loc`. This is enforced by the type signatures of the three interface functions. Located references can be implemented efficiently without attaching any location information to them at runtime in a tagless manner.

Programmers can define user-defined data types with polymorphic locations. For example, one can define a polymorphic location model type by abstracting the location `server` of the reference type in Line 2:

```
data Model = {l}. Content String (List [TodoItem])
                  (Ref l [List [TodoItem]])
```

Accordingly, `init`, `view`, and `update` can be rewritten to use location-parametric models:

- `init : {l}. Model {l}`
- `view : {l}. Model {l} -client-> Html [Msg]`
- `update : {l}.Msg-client->Model{l}-client->Model{l}`

Then one can write a polymorphic page value

```
page : {l}.Page [(Model {l}) Msg] =
      Page (init {l}) (view {l}) (update {l})
```

where `page{client}` is a client only TodoMVC program while `page{server}` is a multi-tier TodoMVC program that behaves like our original example.

3 THE POLYMORPHIC RPC CALCULUS

This section reminds the reader of the polymorphic RPC calculus [5]. It is a polymorphically typed call-by-value λ-calculus with location annotations on λ-abstractions specifying where to run. The calculus offers the notion of polymorphic location to write polymorphically located functions succinctly, which is convenient for programmers.

3.1 The Syntax and the Semantics

Figure 4 shows the syntax and semantics of the polymorphic RPC calculus, $\lambda^{\forall}_{rpc}$ that allows programmers to use the same syntax of λ-application for both local and remote calls, and allows them to compose differently located functions arbitrarily. An important feature is the notion of location variable l for which a location constant a can be substituted. A syntactic object Loc is either a location constant or a location variable. Assuming the client-server model in the calculus, location constants are either **c** denoting client or **s** denoting server.

In the syntax, M denotes terms, and V denotes values. Every λ-abstraction $\lambda^{Loc}x.M$ has a location annotation of Loc. By substituting a location b for a location variable annotation, $(\lambda^{l}x.M)\{b/l\}$ becomes a monomorphic λ-abstraction $\lambda^{b}x.(M\{b/l\})$. This location variable is abstracted by the location abstraction construct $\Lambda l.V$, and it is instantiated by the location application construct $M[Loc]$. Term applications are denoted by $L\ M$. The rest of the syntax are straightforward.

The semantics of $\lambda^{\forall}_{rpc}$ is defined in the style of a big-step operational semantics whose evaluation judgments, $M \Downarrow_a V$, denote that a term M evaluates to a value V at location a. In the semantics,

[1] https://github.com/kwanghoon/todomvc

```
data TodoItem = TodoItem String Bool;
data Model = Content String (List [TodoItem]) (Ref {server} [List [TodoItem]]);
data Msg = Update String | Submit | Toggle Int | Delete Int ;

showItem : TodoItem -client-> Int -client-> Html [Msg]
  = \item @ client idx @ client. case item { TodoItem content done =>
        Element "li" []
         [ Element "input" [ Attribute "type" "checkbox", onClick (Toggle idx)
                  , Property "checked" (if done then "false" else "true") ] []
         , Element "label" [] [ Txt content ]
         , Element "button" [ onClick (Delete idx) ] [ Txt "X" ]   ] };
showList : List [TodoItem] -client-> Html [Msg]
  = \items@client. Element "ul" [] (mapWithCount {client} 0 showItem items);
header : String -client-> Html [Msg]
  = \str @ client. Element "input"
          [ Attribute "placeholder" "What needs to be done?"
          , Property "value" str, onInput Update, onEnter Submit ] [];
view : Model -client-> Html [Msg]
  = \m @ client. case m { Content str visibleList ref =>
          Element "div" [] [ header str, showList visibleList ] };
toggleItem: {l}. TodoItem -l-> TodoItem
  = {l}. \ti @ l. case ti { TodoItem content done =>
        TodoItem content (not {l} done) };
update : Msg -client-> Model -client-> Model
  = \msg @ client model @ client.
     case model { Content line visibleList ref =>
       case msg {
         Update str => Content str visibleList ref;
         Submit => let { u : Unit =
             ref := {server} ( TodoItem line False :: ! {server} ref )
           } Content "" (! {server} ref) ref end ;
         Toggle idx => let { u : Unit =
             ref := {server} (mapOnIndex {server} idx (toggleItem {server})
                              (! {server} ref))
           } Content line (! {server} ref) ref end ;
         Delete idx => let { u : Unit =
             ref := {server} (delete {server} idx (! {server} ref))
           } Content line (! {server} ref) ref end
     }};
serverModel : Ref {server} [List [TodoItem]] = ref {server} Nil;
init : Model = Content "" Nil serverModel;
main : Page [Model Msg] = Page init view update
```

Figure 3: A Multi-tier TodoMVC Program

location annotated λ-abstractions, type abstractions, and location abstractions are all values. So, (Abs), (Tabs), and (Labs) are straightforwardly defined as an identity evaluation relation over them. (App) defines local calls when $a = b$ and remote calls when $a \neq b$ in the same syntax of lambda applications. The evaluation of an application $L\ M$ at location a performs β-reduction at location b, where a λ-abstraction $\lambda^b x.N$ from L has as an annotation, with a value W from M, and it continues to evaluate the β-reduced term $N\{W/x\}$, which is a substitution of W for x in N, at the same location. The remaining semantics rules are easily understood.

As a running example, let us consider a simple term:

$$(\Lambda l.\lambda^l g.g\ 1)[\mathbf{s}]\ (\lambda^{\mathbf{c}} x.x) \quad \text{where } g \text{ has type } Int \xrightarrow{\mathbf{c}} Int$$

Evaluation starting at client goes to server by $(\lambda^{\mathbf{s}} g.g\ 1)\ (\lambda^{\mathbf{c}} x.x)$ and then to the client by $(\lambda^{\mathbf{c}} x.x)\ 1$ resulting in 1 there. The result comes back to the server and then to the client, ending the evaluation.

3.2 The Type System

Figure 5 shows a type system for the polymorphic RPC calculus [5] that can identify remote procedure calls at the type level, supporting location polymorphism. The type language allows function types $A \xrightarrow{Loc} B$. Then every λ-abstraction at unknown location gets assigned $A \xrightarrow{l} B$ using some location variable l. A universal quantifier over a location variable, $\forall l.A$, is also introduced to allow to abstract such occurrences of location variables.

Typing judgments are in the form of $\Gamma \vdash_{Loc} M : A$, saying a term M at location a has type A under a type environment Γ. The location annotation, Loc, is either a location variable or constant. Typing environments Γ have location variables, type variables, and types of variables, as $\{l_1, \cdots, l_n, \alpha_1, \cdots, \alpha_k, x_1 : A_1, \cdots, x_m : A_m\}$.

The typing rules for the polymorphic RPC calculus are defined as follows. (T-App) is a refinement of the conventional typing rule for λ-applications with respect to the combinations of location Loc (where to evaluate the application) and location Loc' (where to evaluate the function). For example, (T-App) is applied to $((\Lambda l.\lambda^l g.g\ 1)[\mathbf{s}])\ (\lambda^{\mathbf{c}} x.x)$ with $Loc = \mathbf{c}$ and $Loc' = \mathbf{s}$, meaning that this application is a remote procedure call from client to server. (T-App) is also applied to $g\ 1$ with $Loc = l$ and $Loc' = \mathbf{c}$. We have to check over l at runtime to make a decision whether or not this is a local procedure call. The other typing rules are explained in the extended version [6].

Syntax

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

Semantics

$$\text{(Abs)}\ \frac{}{\lambda^b x.M \Downarrow_a \lambda^b x.M}$$

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

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

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

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

Figure 4: The semantics for $\lambda^{\forall}_{rpc}$

Types

Type	A, B, C	::=	$base \mid A \xrightarrow{Loc} B \mid \alpha \mid A \times B \mid \forall\alpha.A \mid \forall l.A$
Type env.	Γ	::=	$\emptyset \mid \Gamma, x : A \mid \Gamma, \alpha \mid \Gamma, l$

Typing Rules

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

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

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

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

$$\text{(T-Pair)}\ \frac{\Gamma \vdash_{Loc} L : A \quad \Gamma \vdash_{Loc} M : B}{\Gamma \vdash_{Loc} (L, M) : A \times B}$$

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

Figure 5: The type system for $\lambda^{\forall}_{rpc}$

The type soundness of the type system for the polymorphic RPC calculus, which was formulated as Theorem 3.1 and was proved by [5], guarantees that every remote procedure call thus identified statically will never change to a local procedure call under evaluation. This enables compilers to generate call instructions for local calls and network communication for remote calls safely even though both are in the same syntax of lambda applications.

Theorem 3.1 (Type soundness for $\lambda^{\forall}_{rpc}$ [5]). *For a closed term M, if $\emptyset \vdash_a M : A$ and $M \Downarrow_a V$, then $\emptyset \vdash_a V : A$.*

3.3 The Static Approach to Polymorphic Locations

When a polymorphic application is written in the way that the location of the application, Loc, and the location of the function to run, Loc', may be location variables, compilers cannot statically determine if the lambda application is for remote calls, local calls, or both. The existing slicing compilation method for the typed RPC calculus [4], that is the simply typed and monomorphic subset of the polymorphic RPC calculus [5], cannot deal with such a polymorphic lambda application any more.

The previous study [5] overcame this limitation by translating all polymorphic locations in RPC programs into monomorphic ones by the so called *monomorphisation* translation. This approach is called static because all polymorphic locations can now be resolved at compile-time.

As stated by the study, in the worst case the monomorphisation translation can potentially lead to code explosion by generating client and server versions for each location abstraction. When there are n location abstractions nested subsequently, 2^n monomorphic versions could be generated. This is called a code explosion problem of the static approach to the implementation of the polymorphic RPC calculus.

To show the code explosion problem in the worst case that the study [5] mentioned, let us consider a small example of `S` and `K` combinators written in PolyRPC to make an identity function.

```
1 s : {l1 l2 l3}. [a b c].
2       ((a-l1->b-l1->c) -l3-> (a-l2->b) -l3-> a-l3->c)
3   = {l1 l2 l3}. \f @ l3  g @ l3  x @ l3. f x (g x) ;
4 k : {l}. [a b]. (a -l-> b -l-> a)
5   = {l}. \x @ l  y @ l . x ;
6 identity : {l}. [a]. (a -l-> a)
7   = {l}. \x @ l. s {l l l} (k {l}) (k {l}) x;
8 main : Int  = identity {client} 123
```

Let us call this a *spine location style* SKI program where every multiple-argument function is applied to all its arguments at the same location. There are several variants including a full freedom SKI program by allowing applying a multiple-argument function to each argument all at different locations. In the full freedom program, the `S` combinator will have a location abstraction with six location variables as `{l11 l12 l2 l31 l32 l33}` by replacing the two occurrences of `l1` by `l11` and `l12` and by replacing the three occurrences of `l3` by `l31`, `l32`, and `l33`.

Here is a simple experimental result with these two programs for code size and location checks. By counting the nodes of a program tree (excluding type nodes), the spine location style SKI program is of size 59, and the full freedom SKI program is of size 68. After applying the monomorphisation, the sizes become 190 and 844. Running each of the two programs applies functions 9 times. Both of the spine location style and full freedom SKI programs do dynamic location checks 3 times.

With a preliminary experience with programming PolyRPC, we are not so sure how often the worst case behavior would appear in practice by nested location abstractions as the existing study [5] is concerned. For now, this multi-tier TodoMVC program is the largest program about 300 lines written in PolyRPC. It is of size 1855, increasing up to 2554 after the monomorphisation. Some functions may have nested location abstractions naturally. Consider a located

thread creation function, `fork : {l1 l2}. (ProcId-l1->void) -l2-> ProcId` where `l1` is the location of a child process and `l2` is the location of the parent process. Running `fork {client server}` in the server would create a client process with a parent process id and it would return an id for communication with the client child process. We would need more programming experience, which is left as a future work.

In the next section, we will introduce a new polymorphically typed client-server calculus. Basically, this new CS calculus will be based on statically resolved location as done in the typed RPC calculus [4]. In addition, it will also support a dynamic approach offering a way to determine dynamically whether polymorphic-location lambda applications are local or remote procedure calls. We are interested in the dynamic approach for several reasons. First, the dynamic approach does not have to do any static translations for polymorphic locations at compile-time. It can allow compilers to deal with polymorphic location programs directly. Second, this approach can handle the worst case behavior of the static approach in case such a bad situation happens. In this respect, the dynamic approach can be viewed as a generalization of the static approach. Third, having the dynamic approach itself is of our theoretical interest as a complementary technology. Fortunately, it is found out that it is easy to add the dynamic approach to the portion of the calculus that uses statically resolved locations.

4 A POLYMORPHIC CLIENT-SERVER CALCULUS

Slicing compilation is a desirable feature of multi-tier programming languages because it can slim down code sizes as small as necessary at each location and it can avoid potential security leaks resulting from the server code being available at the client.

The idea of slicing compilation of the polymorphic RPC calculus naturally leads to the introduction of a client-server (CS) calculus where there are two separate programs, one for the client and the other for the server. Client-server programs can be modeled as a pair of client and server function maps, written as $(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$ where Φ maps function names into the codes available at each location. Then the slicing compilation is a translation of RPC calculi terms into pairs of the function maps.

The behavior of the client-server programs will be described using configurations, $Conf$, that are snapshots in the client-server model, written as $\langle client \mid server \rangle$. Firstly, a locally well-formed behavior is required in the client-server programs: the client part is only allowed to look up the client function map to find and run client functions and so is the server part with its own function map. For example, a closure whose function name refers to the server can appear at the client, but an attempt to run the closure at the client would get stuck. Secondly, the client and the server should keep a simple communication protocol: when one attempts to send something, the other should be ready to receive it, and subsequently the roles are changed.

We design a new typed calculus, named a polymorphic client-server (CS) calculus, $\lambda_{cs}^{\forall}$, that serves as a target language where the type system guarantees both the locally well-formed behavior and the simple client-server communication protocol. This is contrasted with the existing CS calculi left in an untyped setting [4, 7].

As in the existing CS calculi, the polymorphic CS calculus syntactically distinguishes local procedure calls, written as $V(W)$, from the remote procedure calls, $\mathsf{req}(V, W)$ and $\mathsf{call}(V, W)$. For example, $\mathsf{req}(V, W)$ is interpreted as sending a pair of V and W to the server and doing a local procedure call there by $V(W)$, and $\mathsf{call}(V, W)$ has the same interpretation but for the reverse direction. The polymorphic CS calculus will use these three syntactic forms of procedure calls wherever location information is statically known, as in the static approach.

In addition, $\lambda_{cs}^{\forall}$ supports a new dynamic approach where location information can be examined in runtime. We introduce a new syntactic form of procedure calls, $\mathsf{gen}(Loc, V, W)$. The semantics of this generic procedure call is to take a location argument Loc, which is the location of the function V. Suppose the client program has such a generic procedure call. Then it becomes a local procedure call if Loc is $\mathbf{c}$, and it becomes a remote one if Loc is $\mathbf{s}$. In the server program, it will have the opposite role. At compile-time, however, either Loc or the current location (or both) may be unknown if they are a location variable. This is where generic procedure calls are necessary to postpone the decision until runtime. The introduction of the generic procedure calls does not break the static approach still ensuring both the locally well-formed behavior and the simple client-server communication protocol.

There are subtle issues in typing the symmetric RPC communication pattern between the client and the server. The symmetric pattern means that before a remote procedure call finishes to get a result, another remote procedure call in the reverse direction can intervene. For example, consider $(\lambda^{\mathbf{s}} g.(g\ 1))\ (\lambda^{\mathbf{c}} x.x)$. Before calling the server function from the client finishes, the client function is called back from the server through g. The existing CS calculi have implemented this RPC communication pattern using *trampolined style* [13] where a single "scheduler" loop, called trampoline, manages all transfers of control by remote procedure call. Whenever a computation performs a unit of work followed by remote procedure call, the remaining work is returned to the scheduler. Thus every remote procedure call is wrapped with a trampoline loop that would repeat to process intervening procedure call requests from the other location and that would eventually stop on a result of the remote procedure call.

A type system for the polymorphic CS calculus is designed to have the calculus be high-level so that the low-level details of the trampoline-based implementation are not explicitly exposed to the syntactic terms and types. For this, monads are used to abstract the trampoline details of remote procedure calls and to focus on their eventual result values. Every term of monad type $T\ A$ may involve remote procedure calls that will eventually return a value of type A. Roughly speaking, all of $\mathsf{req}(V, W)$, $\mathsf{call}(V, W)$, and $\mathsf{gen}(Loc, V, W)$ will get assigned this type whenever V is a function of type of A to B and W is of type A. Thus typing remote procedure calls becomes as simple as typing local procedure calls, which is an advantage of our design decision. Typing the simple client-server communication protocol is going to be simple as will be explained in the following.

The other issue is about typing concurrency between the client and the server in the polymorphic CS calculus. The RPC calculi are high-level enough to be able to treat them like a sequential calculus over the client and the server. Once communication primitives, such

Types & Terms

Type	A, B, C	::=	$base \mid A \xrightarrow{Loc} B \mid Clo(A) \mid A \times B \mid \alpha \mid \forall\alpha.A \mid \forall l.A \mid T\ A$
Term	L, M, N	::=	$V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \pi_i(V) \mid V(W) \mid V[A] \mid V[Loc]$
Value	V, W	::=	$x \mid (V, W) \mid clo(\overline{W}, F) \mid \Lambda\alpha.V$
		|	$\mathsf{unit}\ V \mid \mathsf{do}\ x \leftarrow M\ \mathsf{in}\ N \mid \mathsf{req}(V, W) \mid \mathsf{call}(V, W) \mid \mathsf{gen}(Loc, V, W)$
Code term	$Code$	::=	$\overline{l}\ \overline{\alpha}.\ \overline{z}.\ OpenCode$ where $OpenCode ::= \lambda x.M \mid \Lambda l.V$
Code name	F	::=	$F_{name}[\overline{Loc}\ \overline{A}]$
Program	prg	::=	$(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$
Function map	Φ	::=	$\{\ F_{name,1} : Ty_1 = Code_1,\ \cdots,\ F_{name,n} : Ty_n = Code_n\ \}$ where Ty is $\overline{l}\ \overline{\alpha}.\ \overline{B}.A$

Figure 6: Types and terms in the polymorphic CS calculus

as *send* and *receive*, were introduced to the low-level implementation of the RPC calculus, some very limited form of concurrency would appear: a sender would have to be ready before a receiver would be so, or vice versa. To deal with this, we could resort to some advanced techniques, such as session types, but we decide to remain in a simpler type system.

Here is a summary of a few highlights of the polymorphic CS calculus. First, the type system guarantees that while functions may be passed to arbitrary locations, every function must run at the declared location. Second, it is designed to support the combination of the static approach relying on monomorphisation and a complementary dynamic approach for handling polymorphic-location programs directly. Third, it employs a monadic abstraction for *trampolined style* RPC communication that allows us to treat the polymorphic CS calculus like a sequential calculus over the client and server.

4.1 Types and Terms

The polymorphic CS calculus is as shown in Figure 6. A monadic type, $T\ A$, denotes a computation that produces values of type A and may involve remote procedure calls during the computation. A term for unit operation, $\mathsf{unit}\ V$, turns values into monadic ones, and a term for monad composition, $\mathsf{do}\ x \leftarrow M\ \mathsf{in}\ N$, transforms monadic values of type $T\ A$ from M into other monadic values of type $T\ B$ from N after binding the unwrapped value of type A to the variable x. Three monad terms are introduced for remote and generic procedure calls: $\mathsf{req}(V, W)$, $\mathsf{call}(V, W)$, and $\mathsf{gen}(Loc, V, W)$. We call these five terms *monadic values*. The others are called *plain terms* and *values*.

As well as the monadic types, closure types, $Clo(A)$, are introduced for typing closures with functions or location abstractions hiding free variables occurring in them. Closures are allowed to be passed over tiers. In addition, all kinds of types and terms in the polymorphic RPC calculus are adopted.

Every program in $\lambda^{\forall}_{cs}$ is a pair of client and server function maps, $(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$. Function maps Φ are defined as mappings of names, F_{name}, into pairs of closed types, Ty, and closed codes, $Code$. Such a mapping is described as $F_{name} : Ty = Code$.

Every closed code is written as $\overline{l}\ \overline{\alpha}.\overline{z}.OpenCode$. The prefix denotes abstractions over location variables, type variables, and free variables occurring in the open code. Values for the free variables in the open code are stored in closures. Types for the free type variables are not stored in closures but they replace the occurrences of the free type variables in the open code. Locations for the free location variables are treated as the same as types. The name of code, F, in a closure is defined as a name with location and type applications, $F_{name}[\overline{Loc}\ \overline{A}]$, which represents an instance of a closed code referred by F_{name}, $\overline{z}.(OpenCode\{\overline{Loc}\ \overline{A}/\overline{l}\ \overline{\alpha}\})$, where the occurrences of location variables and type variables are replaced by the locations and types, respectively. Each open code denoted by $OpenCode$ is defined as either a lambda abstraction or a location abstraction. Because type abstractions will be erased later in the type erasure semantics, we will never construct any closures for type abstractions treating them as a value, not as open code.

From the running example in Section 3.1, one can obtain a $\lambda^{\forall}_{cs}$ program as: $main = \mathsf{do}\ \ h\ \leftarrow clo(\emptyset, f_1)[\mathbf{s}]\ \mathsf{in}\ \mathsf{req}(\ h,\ clo(\emptyset, f_3)\ )$ where $(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$ is

$$
\begin{array}{lcll}
f_1 : \emptyset.\emptyset.A_1 & = & \emptyset.\emptyset.\ \Lambda l.\ \mathsf{unit}\ (clo(\emptyset, f_2[l])) & \in\ \ \Phi_{\mathbf{c}}, \Phi_{\mathbf{s}} \\
f_2 : l.\emptyset.A_2 & = & l.\emptyset.\ \lambda g.\ \mathsf{gen}(\ \mathbf{c},\ g,\ 1\ ) & \in\ \ \Phi_{\mathbf{c}}, \Phi_{\mathbf{s}} \\
f_3 : \emptyset.\emptyset.A_3 & = & \emptyset.\emptyset.\ \lambda x.\ \mathsf{unit}\ x & \in\ \ \Phi_{\mathbf{c}}
\end{array}
$$

such that $A_1 = \forall l.T\ Clo(A_2)$, $A_2 = Clo(A_3) \xrightarrow{l} T\ Int$, and $A_3 = Int \xrightarrow{\mathbf{c}} T\ Int$. Note that the empty sequence is denoted by $\emptyset$.

4.2 A Type System

The purpose of the type system for the polymorphic CS calculus is to guarantee both locally well-formed behavior and a simple client-server communication protocol. As previously, typing judgments for terms are $\Gamma \vdash_{Loc} M : A$ saying a term M has type A at location Loc under a type environment Γ.

Every client-server program, $(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$ is well-formed if there exist $\Phi^{\circ}_{\mathbf{c}}$, $\Phi^{\circ}_{\mathbf{s}}$, and Φ_{common} such that all function names are distinct, $\Phi_{\mathbf{c}} \cup \Phi_{\mathbf{s}} = \Phi^{\circ}_{\mathbf{c}} \uplus \Phi^{\circ}_{\mathbf{s}} \uplus \Phi_{common}$, and each binding $(F_{name} : \overline{l}\ \overline{\alpha}.\overline{B}.A = \overline{l}\ \overline{\alpha}.\overline{z}.OpenCode)$ where $\Gamma = \{\overline{l}, \overline{\alpha}, \overline{z : B}\}$, satisfies either

- $A = A_1 \xrightarrow{Loc} A_2$, $OpenCode = \lambda x.M$, $\Gamma, x : A_1 \vdash_{Loc} M : A_2$, and the binding is in Φ_{Loc}; or
- $A = \forall l.A_1$, $OpenCode = \forall l.V$, $\Gamma, l \vdash_{Loc} V : A_1$ for all arbitrary locations Loc, and the binding is in Φ_{common}.

where for notation, $\Phi_{Loc} = \Phi^{\circ}_{a}$ if $Loc = a$, and otherwise, if Loc is a location variable, it is Φ_{common}.

Intuitively, every client-server program is type-checked under the client and server function maps. The union of the two functions maps has to be decomposed into three disjoint ones: Φ_{common}, $\Phi^{\circ}_{\mathbf{c}}$,

Typing Rules

$$\text{(T-Var)}\ \frac{\Gamma(x) = A}{\Gamma \vdash_{Loc} x : A} \quad \text{(T-Let)}\ \frac{\Gamma \vdash_{Loc} M : A \quad \Gamma, x : A \vdash_{Loc} N : B}{\Gamma \vdash_{Loc} \mathsf{let}\ x = M\ \mathsf{in}\ N : B} \quad \text{(T-Clo)}\ \frac{\vdash F : \overline{B_i}.A \quad \emptyset \vdash_{Loc} \overline{W_i} : \overline{B_i}}{\Gamma \vdash_{Loc} clo(\overline{W_i}, F) : Clo(A)}$$

$$\text{(T-Proj-i)}\ \frac{\Gamma \vdash_{Loc} V : A_1 \times A_2 \quad i \in \{1,2\}}{\Gamma \vdash_{Loc} \pi_i(V) : A_i} \quad \text{(T-Pair)}\ \frac{\Gamma \vdash_{Loc} V : A \quad \Gamma \vdash_{Loc} W : B}{\Gamma \vdash_{Loc} (V, W) : A \times B}$$

$$\text{(T-Tabs)}\ \frac{\Gamma, \alpha \vdash_{Loc_0} V : A\ \text{ for all } Loc_0}{\Gamma \vdash_{Loc} \Lambda\alpha.V : \forall\alpha.A} \quad \text{(T-Tapp)}\ \frac{\Gamma \vdash_{Loc} V : \forall\alpha.A \quad relocatable(B)}{\Gamma \vdash_{Loc} V[B] : A\{B/\alpha\}} \quad \text{(T-Lapp)}\ \frac{\Gamma \vdash_{Loc} V : Clo(\forall l.A)}{\Gamma \vdash_{Loc} V[Loc'] : A\{Loc'/l\}}$$

$$\text{(T-Unit)}\ \frac{\Gamma \vdash_{Loc} V : A \quad relocatable(A)}{\Gamma \vdash_{Loc} \mathsf{unit}\ V : TA} \quad \text{(T-Bind)}\ \frac{\Gamma \vdash_{Loc} M : TA \quad \Gamma, x : A \vdash_{Loc} N : TB}{\Gamma \vdash_{Loc} \mathsf{do}\ x \leftarrow M\ \mathsf{in}\ N : TB}$$

$$\text{(T-App)}\ \frac{\Gamma \vdash_{Loc} V : Clo(A \xrightarrow{Loc} B) \quad \Gamma \vdash_{Loc} W : A}{\Gamma \vdash_{Loc} V(W) : B} \quad \text{(T-Gen)}\ \frac{\Gamma \vdash_{Loc} V : Clo(A \xrightarrow{Loc'} TB) \quad \Gamma \vdash_{Loc} W : A \quad relocatable(A)}{\Gamma \vdash_{Loc} \mathsf{gen}(Loc', V, W) : TB}$$

$$\text{(T-Req)}\ \frac{\Gamma \vdash_{\mathbf{c}} V : Clo(A \xrightarrow{\mathbf{s}} TB) \quad \Gamma \vdash_{\mathbf{c}} W : A \quad relocatable(A)}{\Gamma \vdash_{\mathbf{c}} \mathsf{req}(V, W) : TB} \quad \text{(T-Call)}\ \frac{\Gamma \vdash_{\mathbf{s}} V : Clo(A \xrightarrow{\mathbf{c}} TB) \quad \Gamma \vdash_{\mathbf{s}} W : A \quad relocatable(A)}{\Gamma \vdash_{\mathbf{s}} \mathsf{call}(V, W) : TB}$$

Typing rule for function names

$$\text{(T-F-Abs)}\ \frac{(F_{name} : \overline{l}\ \overline{\alpha}.\ \overline{B}.A \xrightarrow{Loc'} B = \overline{l}\ \overline{\alpha}.\overline{z}.\lambda x.M) \in \Phi_{Loc'} \quad \overline{l}\ \overline{\alpha}, \overline{z : B}, x : A \vdash_{Loc'} M : B \quad \overline{relocatable(A_i)} \quad \overline{relocatable(B_j)}}{\vdash F_{name}[\overline{Loc}\ \overline{A}] : \overline{B\{\{\overline{Loc}\ \overline{A}/\overline{l}\ \overline{\alpha}\}\}}.(A \xrightarrow{Loc'} B)\{\{\overline{Loc}\ \overline{A}/\overline{l}\ \overline{\alpha}\}\}}$$

$$\text{(T-F-LAbs)}\ \frac{(F_{name} : \overline{l}\ \overline{\alpha}.\ \overline{B}.\forall l.A = \overline{l}\ \overline{\alpha}.\overline{z}.\Lambda l.V) \in \Phi_{common} \quad \overline{l}\ \overline{\alpha}, \overline{z : B}, l \vdash_{Loc_0} V : A\ \text{ for all } Loc_0 \quad \overline{relocatable(A_i)} \quad \overline{relocatable(B_j)}}{\vdash F_{name}[\overline{Loc}\ \overline{A}] : \overline{B\{\{\overline{Loc}\ \overline{A}/\overline{l}\ \overline{\alpha}\}\}}.(\forall l.A)\{\{\overline{Loc}\ \overline{A}/\overline{l}\ \overline{\alpha}\}\}}$$

Figure 7: The type system for the polymorphic CS calculus

and $\Phi^{\circ}_{\mathbf{s}}$. When an open code is a function whose type has a constant location annotation a, its binding belongs to Φ°_{a}. When an open code is associated with a location variable annotation or it is a location abstraction, its binding belongs to the common function map, Φ_{common}. When such a decomposition is possible using the typing rules, the client-server program is said to be well-formed.

The typing rules in Figure 7 are designed to guarantee the locally well-formed behavior. Every remote procedure call has one's own location: $\mathsf{req}(V, W)$ is a server procedure call from the client. (T-Req) specifies $\mathbf{c}$ as the location for evaluation and describes the procedure V as a closure with a server function of type $Clo(A \xrightarrow{\mathbf{s}} TB)$. The return type TB denotes that the result values of the remote procedure call are of type B involving a trampoline communication. (T-Call) is defined similarly for the reverse direction call, $\mathsf{call}(V, W)$.

Given a remote procedure call at the client with a typing derivation concluding $\emptyset \vdash_{\mathbf{c}} \mathsf{req}(V, W) : TB$, we are able to construct another typing derivation now for a local procedure call at the server concluding with $\emptyset \vdash_{\mathbf{s}} V(W) : TB$ as long as the two values V and W are relocatable, i.e., $\emptyset \vdash_{\mathbf{c}} V : Clo(A \xrightarrow{\mathbf{s}} TB)$ implies $\emptyset \vdash_{\mathbf{s}} V : Clo(A \xrightarrow{\mathbf{s}} TB)$ and $\emptyset \vdash_{\mathbf{c}} W : A$ implies $\emptyset \vdash_{\mathbf{s}} W : A$.

In fact, plain values can be shown to be all relocatable. To capture relocatable values, we define a predicate over types, $relocatable(A)$. If A is one of α, $base$, $Clo(B)$, and $\forall\alpha.B$, then A is relocatable. If both A and B are relocatable, then $A \times B$ is relocatable too. Otherwise, $relocatable(T\ A)$ is false for any A. In other words, integers are relocatable. Every closure can be located at the client or the server regardless of the function location.

In (T-Req) and (T-Call), V has closure type, so it is relocatable. For W, the typing rules enforce it to be relocatable by the extra condition. In an ill-formed term, $\mathsf{req}(V, \mathsf{req}(W_1, W_2))$, when sent to to the server, one could attempt to invoke $\mathsf{req}(W_1, W_2)$ at the server violating the well-formed local behavior. This is prevented by the predicate. This completes a justification for an interplay between typing remote and local procedure calls.

In (T-Unit), the type of V in $\mathsf{unit}\ V$ is defined as relocatable because this term is used to return one location to the other. (T-Bind) is straightforward.

In (T-Tapp), the argument type B of type applications $V[B]$ is relocatable since type variables are defined as relocatable by the predicate. The system is currently limited in that type abstractions can only be instantiated with relocatable types, but this is not a problem in practice because we translate all types actually occurring as type applications in the source language to relocatable types anyway. This limitation is due to the fact that we otherwise have no way of determining whether a type variable α is relocatable, for example in (T-Gen), (T-Call) or (T-Req); the solution is to ensure that the types eventually substituted for type variable are always relocatable. It would be interesting to remove this restriction, for example using qualified types [15] to track which type variables actually need to be relocatable.

(T-Clo) is a typing rule for closures. It uses one of the typing rules (T-F-Abs) and (T-F-Labs) for two purposes. One is for clarifying

$$
\begin{array}{llll}
\text{Eval. context} & E[\,] & ::= & E_{let}[\,] \mid \mathsf{do}\ x \leftarrow E[\,]\ \mathsf{in}\ M \\
 & E_{let}[\,] & ::= & [\,] \mid \mathsf{let}\ x = E_{let}[\,]\ \mathsf{in}\ M \\
\text{Stack} & \Delta & ::= & \epsilon \mid E[\,];\Delta \\
\text{Configuration} & Conf & ::= & \langle M;\Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle \mid \langle \Delta_{\mathbf{c}} \mid M;\Delta_{\mathbf{s}}\rangle \\
\text{Conf. context} & \Sigma & ::= & \langle E[\,];\Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle \mid \langle \Delta_{\mathbf{c}} \mid E[\,];\Delta_{\mathbf{s}}\rangle
\end{array}
$$

Figure 8: The runtime system and contexts

which function map the type checker should look at by a similar idea to that used in the function map decomposition. The other is for getting an instantiated type by appropriate location and type applications. The typing rules also enforce that all type arguments and all free type variables are relocatable.

(T-Tabs) prohibits location-dependent values from $\Lambda\alpha.V$ by having a condition of typing V at arbitrary locations. For example, $\Lambda\alpha.\mathsf{req}(f, arg)$ is ill-typed because the monadic value $\mathsf{req}(f, arg)$ is well-typed only at client by (T-Req).

The other typing rules, (T-Let), (T-Proj-i), (T-Lapp), (T-Var), and (T-Pair) are straightforward except (T-Gen).

Until now, the typing rules have aimed at ensuring the well-formed behavior by the static approach while (T-Gen) is for the dynamic approach. The typing rule for $\mathsf{gen}(Loc', V, W)$ specifies that Loc' should be the function location. Thus the function location in the closure type $Clo(A \xrightarrow{Loc'} TB)$ becomes available in the term-level for examination against the evaluation location, Loc, in runtime. The type soundness property shows that the introduction of generic procedure calls and (T-Gen) preserves the statically resolved location information by the static approach. Note that (T-App), (T-Req) and (T-Call) can be viewed as specific instances of (T-Gen), demanding no runtime location examination.

In the running example, a code of function name f_2

$$f_2 : l.\emptyset.A_2 \ = \ l.\emptyset.\ \lambda g.\ \mathsf{gen}(\ \mathbf{c},\ g,\ 1\ ) \ \in \ \Phi_{\mathbf{c}}, \Phi_{\mathbf{s}}$$

would be type-checked by the following typing judgment

$$l, g : Clo(Int \xrightarrow{\mathbf{c}} T\ Int) \vdash_l \mathsf{gen}(\mathbf{c}, g, 1) : T\ Int$$

where $A_2 = Clo(Int \xrightarrow{\mathbf{c}} T\ Int) \xrightarrow{l} T\ Int$.

In the next section, we will discuss the well-formed communication of the polymorphic CS calculus and will explain how our *stack typing* can guarantee the simple client-server protocol.

4.3 Runtime Contexts and Typing Rules

The communication aspect of $\lambda^{\forall}_{cs}$ involves runtime components and associated contexts as introduced in Figure 8. Configurations, $Conf$, are snapshots in the client-server model, written as $\langle client \mid server\rangle$. There are two kinds of configurations: $\langle M_{\mathbf{c}};\Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$ describes when the client evaluates a term $M_{\mathbf{c}}$ on the client stack $\Delta_{\mathbf{c}}$ and the server stack $\Delta_{\mathbf{s}}$, and $\langle \Delta_{\mathbf{c}} \mid M_{\mathbf{s}};\Delta_{\mathbf{s}}\rangle$ is for the reverse roles of the client and the server. Stacks are defined as sequences of evaluation contexts separated by semicolons, $E_1[\,];\cdots;E_n[\,]$, and the empty stack is denoted by ϵ. Stacks increase on each remote procedure call, and they decrease on its return. Each evaluation context in a stack denotes a term with a hole waiting for return values from the remote procedure call.

Stack typing rules

$$\text{(T-Stk-Empty)}\ \frac{}{\vdash_a \epsilon \mid \epsilon : A \Rightarrow A}$$

$$\text{(T-Stk-Client)}\ \frac{x : A \vdash_{\mathbf{s}} E[x] : TC \qquad \vdash_{\mathbf{s}} \Delta_{\mathbf{c}}|\Delta_{\mathbf{s}} : TC \Rightarrow B}{\vdash_{\mathbf{c}} \Delta_{\mathbf{c}} \mid E[\,];\Delta_{\mathbf{s}} : A \Rightarrow B}$$

$$\text{(T-Stk-Server)}\ \frac{x : A \vdash_{\mathbf{c}} E[x] : TC \qquad \vdash_{\mathbf{c}} \Delta_{\mathbf{c}}|\Delta_{\mathbf{s}} : TC \Rightarrow B}{\vdash_{\mathbf{s}} E[\,];\Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}} : A \Rightarrow B}$$

Configuration typing rules

$$\text{(T-Client)}\ \frac{\emptyset \vdash_{\mathbf{c}} M : TA \qquad \vdash_{\mathbf{c}} \Delta_{\mathbf{c}}|\Delta_{\mathbf{s}} : TA \Rightarrow B}{\vdash \langle M;\Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle : B}$$

$$\text{(T-Server)}\ \frac{\emptyset \vdash_{\mathbf{s}} M : TA \qquad \vdash_{\mathbf{s}} \Delta_{\mathbf{c}}|\Delta_{\mathbf{s}} : TA \Rightarrow B}{\vdash \langle \Delta_{\mathbf{c}} \mid M;\Delta_{\mathbf{s}}\rangle : B}$$

Figure 9: Stack typing and configuration typing

The simple communication protocol that should be respected by the client and the server is this. When one attempts to send something, the other should be ready to receive it, and after that, the roles should be changed. This should be repeated until the two stacks are empty.

Figure 9 shows typing rules for stacks and configurations. Stack typing judgements $\vdash_a \Delta_{\mathbf{c}}|\Delta_{\mathbf{s}} : A \Rightarrow B$ is read as: the client and server respect the communication protocol by the stacks viewed at the location a cooperatively producing a result value of type B at the location whenever a value of type A at the location is sent to the other. This generalizes the idea that evaluation contexts $E[\,]$ can be understood as a function filling a value of type A in the hole and evaluating the completed term to produce a result of type B

In (T-Stk-Client), whenever a term, $E[x]$, completed with a variable x for a value of type A received from the client has type TC at the server, and the stacks except the evaluation context, $\Delta_{\mathbf{c}}|\Delta_{\mathbf{s}}$, from the server view are well-formed with stack type $TC \Rightarrow B$, the two stacks from the client view will be well-formed with stack type $A \Rightarrow B$. Alternating views in stack typing judgments are changing roles in the communication, i.e., who to send and who to receive. For example, $\langle M; E[] \mid \epsilon\rangle$ is an ill-formed configuration. When the client sends the value of M to the server, there is no one to receive. Also, the client will receive nothing from the server through $E[]$ on the client stack. In (T-Stk-Server), the unwinding begins with sending a value to the client from the server. By (T-Stk-Empty), a pair of the two empty stacks is treated as an identity continuation.

Configurations are well-formed when well-typed terms of type TA fit well-typed pairs of stacks of stack type $TA \Rightarrow B$. So, configuration typing rules combine term typing with stack typing. By (T-Client), we assign a type B to each client-running configuration, $\langle M;\Delta_{\mathbf{c}}|\Delta_{\mathbf{s}}\rangle$, if a closed term M has type TA at the client and the stacks $\Delta_{\mathbf{c}}|\Delta_{\mathbf{s}}$ has type $TA \Rightarrow B$ viewed from the client. By (T-Server), we can define a typing rule for server-running configurations, $\langle \Delta_{\mathbf{c}}|M;\Delta_{\mathbf{s}}\rangle$, in the same manner at the server.

In the running example, a term $(\lambda^{\mathbf{c}}x.x)\ 1$ at client intervening a remote call from client to server would correspond to a configuration $\langle\ clo(\emptyset, f_3)(1); [\,] \mid [\,]\ \rangle$, which is well-formed by

$$\text{(T-Client)}\ \frac{\emptyset \vdash_{\mathbf{c}} clo(\emptyset, f_3)(1) : T\ Int \qquad \vdash_{\mathbf{c}} [\,]|[\,] : T\ Int \Rightarrow T\ Int}{\vdash \langle clo(\emptyset, f_3)(1); [\,] \mid [\,]\rangle : T\ Int}$$

4.4 The Semantics and Type Soundness

The semantics for the polymorphic client-server calculus is described by the small-step operational semantics over configurations, $Conf \longrightarrow Conf'$, as shown in Figure 10. The basic idea is to evaluate terms to monadic values and then to interpret the monadic values to perform remote procedure calls. The semantics is defined by a sequence of configurations whose configuration types are all TA and whose last configuration will normally have a form as $\langle \mathsf{unit}\ V; \epsilon | \epsilon \rangle$ giving a value V of type A.

In the semantics, communication rules manages all transfers of control by remote procedure call while local reduction rules are used to perform each unit of work between two subsequent RPCs. For the local reduction rules, configuration contexts, $\Sigma_a[\ ]$, capture a local reduction at the location a by (E-Local). Then the other local rules are applied. Evaluation contexts have two forms: $E[\ ]$ and $E_{let}[\ ]$. Computational evaluation is captured by $E[\ ]$ while plain term evaluation is captured by $E_{let}[\ ]$. Configuration contexts are either $\Sigma_{\mathbf{c}}[-]$ denoting $\langle E[-]; \Delta_{\mathbf{c}} | \Delta_{\mathbf{s}} \rangle$ or $\Sigma_{\mathbf{s}}[-]$ denoting $\langle \Delta_{\mathbf{c}} | E[-]; \Delta_{\mathbf{s}} \rangle$. (E-App) and (E-LApp) use only the function map at the location a for looking up codes. An ill-formed program could get stuck because of the absence of the code to run.

In the communication rules, (E-Req) and (E-Call) send a function and an argument to the other location leaving an evaluation context on the stack at the current location. The symmetry of the two rules demands a trampolined style implementation that one direction remote call can be intervened by the other one. Later, our implementation of $\lambda^{\forall}_{cs}$ will make trampolined style loops explicit.

The three Unit rules send back the remote procedure call results to the other location. The four Gen rules examine the location in the first argument against the current location of the generic procedure call to determine if the call is local or remote.

The following are evaluation steps for the running example starting with $\langle main \mid \epsilon \rangle$.

$$
\begin{array}{ll}
 & \langle \mathsf{do}\ h \leftarrow clo(\emptyset, f_1)[\mathbf{s}]\ \mathsf{in}\ \mathsf{req}(\ h,\ clo(\emptyset, f_3)\ )\ |\ \epsilon \rangle \\
\longrightarrow & \langle \mathsf{do}\ h \leftarrow \mathsf{unit}\ (clo(\emptyset, f_2[\mathbf{s}]))\ \mathsf{in}\ \mathsf{req}(\ h,\ clo(\emptyset, f_3)\ )\ |\ \epsilon \rangle \\
\longrightarrow & \langle\ \mathsf{req}(\ clo(\emptyset, f_2[\mathbf{s}])\ ,\ clo(\emptyset, f_3)\ )\ |\ \epsilon\ \rangle \\
\longrightarrow & \langle\ [\ ]\ |\ clo(\emptyset, f_2[\mathbf{s}])(\ clo(\emptyset, f_3)\ )\ \rangle \\
\longrightarrow & \langle\ [\ ]\ |\ \mathsf{gen}(\mathbf{c}, clo(\emptyset, f_3), 1)\ \rangle \\
\longrightarrow & \langle\ [\ ]\ |\ \mathsf{call}(\ clo(\emptyset, f_3)\ , 1)\ \rangle \\
\longrightarrow & \langle\ clo(\emptyset, f_3)\ (1); [\ ]\ |\ [\ ]\ \rangle \\
\longrightarrow & \langle\ \mathsf{unit}\ 1; [\ ]\ |\ [\ ]\ \rangle \\
\longrightarrow & \langle\ [\ ]\ |\ \mathsf{unit}\ 1\ \rangle \\
\longrightarrow & \langle\ \mathsf{unit}\ 1\ |\ \epsilon\ \rangle
\end{array}
$$

The type soundness property for the polymorphic CS calculus is proven as Theorem 4.1 by showing the type preservation and the progress properties.

Theorem 4.1 (Type soundness). *Given a well-formed polymorphic CS program $(\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}})$ with the main term M, if $\vdash \langle M; \epsilon | \epsilon \rangle : TA$, either $\langle M; \epsilon | \epsilon \rangle \longrightarrow^* \langle unit\ V; \epsilon | \epsilon \rangle$ or it loops indefinitely.*

4.5 A Typed Slicing Compilation

Our typed slicing compilation translates $\lambda^{\forall}_{rpc}$ into $\lambda^{\forall}_{cs}$. Basically, it is a monadic conversion with a slicing, compiling RPC terms of type A into monadic client and server terms of type T ($\mathcal{V}[\![A]\!]$) denoting a computation of values of type $\mathcal{V}[\![A]\!]$ that may involve calling remote procedures during the computation.

Figure 11 shows the typed slicing compilation rules. It comprises type compilations, $\mathcal{C}[\![A]\!]$ and $\mathcal{V}[\![A]\!]$, and term compilations, $C[\![M]\!]_{\Gamma,Loc,A}$ and $\mathcal{V}[\![V]\!]_{\Gamma,Loc,A}$. The term compilation rules actually take as its input typing derivations for terms, such as typing derivations concluding with typing judgments $\Gamma \vdash_{Loc} M : A$ or $\Gamma \vdash_{Loc} V : A$. The output is two function maps, $\Phi_{\mathbf{c}}$ and $\Phi_{\mathbf{s}}$, with a main client expression. We use a notation, $(F_{name} : Ty = Code) \in \Phi_{Loc}$, for adding the binding of F_{name} to function stores. If Loc in Φ_{Loc} is a location variable, the compilation adds the binding both to the client function map and the server function map.

The type and term compilation rules are quite straightforward and are in line with the ideas explained until now. Both lambda abstractions and location abstractions are compiled as closures while type abstractions are compiled as themselves that will be erased later. Lambda applications can be compiled with the new generic application by default. But by analyzing the location of the lambda applications (Loc) and a function location (Loc'), it is easy to have optimized compilation with local and remote application terms whenever the relevant location information is statically available, as was done for compiling the typed RPC calculus [4]. When $Loc = Loc'$, $f(x)$ can replace $\mathsf{gen}(Loc', f, x)$ in the compilation. When $Loc = \mathbf{c} \wedge Loc' = \mathbf{s}$ and $Loc = \mathbf{s} \wedge Loc' = \mathbf{c}$, $\mathsf{req}(f, x)$ and $\mathsf{call}(f, x)$ will do so, respectively. Location applications and type applications are compiled as themselves but only the latter will be erased later. More explanations are available in the extended version [6].

By definition, the slicing compilation rules guarantee a linear bound on the size of target programs, incurring no code explosion problem like the one by the monomorphisation.

Now we state the type correctness and the semantic correctness properties of the typed slicing compilation rules as follows. By the type correctness property, every well-typed term in the polymorphic RPC calculus will be compiled into a well-typed term in the CS calculus by the typed slicing compilation.

Theorem 4.2 (Type correctness). *If $\Gamma \vdash_{Loc} M : A$ in $\lambda^{\forall}_{rpc}$ then $\mathcal{V}[\![\Gamma]\!] \vdash_{Loc} C[\![M]\!]_{\Gamma,Loc,A} : C[\![A]\!]$ in $\lambda^{\forall}_{cs}$ where $\mathcal{V}[\![\Gamma]\!]$ is a pointwise extension of the type compilation.*

We can also prove the semantic correctness of the slicing typed compilation meaning that whenever a well-typed term evaluates to a value under the semantics of the polymorphic RPC calculus, the compiled term will evaluate to the compiled value.

Theorem 4.3 (Semantic correctness). *If $\emptyset \vdash_{\mathbf{c}} M : A$ and $M \Downarrow_{\mathbf{c}} V$ then $\langle C[\![M]\!]_{\emptyset,\mathbf{c},A} | \epsilon \rangle \longrightarrow^* \langle C[\![V]\!]_{\emptyset,\mathbf{c},A} | \epsilon \rangle$.*

5 IMPLEMENTATION OF THE POLYMORPHIC CS CALCULUS

This section discusses how to implement $\lambda^{\forall}_{cs}$ client and server sliced programs efficiently. Firstly, the programs use types that were necessary for the slicing compilation but are not for execution. In the implementation, we want to erase the types but should retain the locations necessary for runtime examination. Secondly, the notion of monads in $\lambda^{\forall}_{cs}$ was useful before as an abstraction and we now need to implement it using low-level primitives.

[Local reduction]

$$\text{(E-Local)}\quad \frac{M \longrightarrow^a M'}{\Sigma_a[M] \longrightarrow \Sigma_a[M']}$$

(E-Let)	$\mathsf{let}\ x = V\ \mathsf{in}\ M$	$\longrightarrow^a$	$M\{V/x\}$
(E-Do)	$\mathsf{do}\ x \leftarrow \mathsf{unit}\ V\ \mathsf{in}\ M$	$\longrightarrow^a$	$M\{V/x\}$
(E-Proj-i)	$\pi_i(V_1, V_2)$	$\longrightarrow^a$	V_i where $i = 1, 2$
(E-TApp)	$(\Lambda\alpha.V)[A]$	$\longrightarrow^a$	$V\{A/\alpha\}$
(E-App)	$clo(\overline{W}, F)(V)$	$\longrightarrow^a$	$M\{\overline{W}/\overline{z}\}\{V/x\}$ if $\Phi_a(F) = \overline{z}.\lambda x.M$
(E-LApp)	$clo(\overline{W}, F)[b]$	$\longrightarrow^a$	$V\{\overline{W}/\overline{z}\}\{b/l\}$ if $\Phi_a(F) = \overline{z}.\Lambda l.V$

[Communication]

(E-Req)	$\langle E[\mathsf{req}(V,W)]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle E[\,]; \Delta_{\mathbf{c}} \mid V(W); \Delta_{\mathbf{s}}\rangle$
(E-Call)	$\langle \Delta_{\mathbf{c}} \mid E[\mathsf{call}(V,W)]; \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle V(W); \Delta_{\mathbf{c}} \mid E[\,]; \Delta_{\mathbf{s}}\rangle$
(E-Unit-C)	$\langle \mathsf{unit}\ V; \Delta_{\mathbf{c}} \mid E[\,]; \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle \Delta_{\mathbf{c}} \mid E[\mathsf{unit}\ V]; \Delta_{\mathbf{s}}\rangle$
(E-Unit-S)	$\langle E[\,]; \Delta_{\mathbf{c}} \mid \mathsf{unit}\ V; \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle E[\mathsf{unit}\ V]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$
(E-Unit-S-E)	$\langle \epsilon \mid \mathsf{unit}\ V\rangle$	$\longrightarrow$	$\langle \mathsf{unit}\ V \mid \epsilon\rangle$
(E-Gen-C-C)	$\langle E[\mathsf{gen}(\mathbf{c}, V, W)]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle E[V(W)]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$
(E-Gen-S-C)	$\langle E[\mathsf{gen}(\mathbf{s}, V, W)]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle E[\mathsf{req}(V,W)]; \Delta_{\mathbf{c}} \mid \Delta_{\mathbf{s}}\rangle$
(E-Gen-C-S)	$\langle \Delta_{\mathbf{c}} \mid E[\mathsf{gen}(\mathbf{c}, V, W)]; \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle \Delta_{\mathbf{c}} \mid E[\mathsf{call}(V,W)]; \Delta_{\mathbf{s}}\rangle$
(E-Gen-S-S)	$\langle \Delta_{\mathbf{c}} \mid E[\mathsf{gen}(\mathbf{s}, V, W)]; \Delta_{\mathbf{s}}\rangle$	$\longrightarrow$	$\langle \Delta_{\mathbf{c}} \mid E[V(W)]; \Delta_{\mathbf{s}}\rangle$

Figure 10: The semantics for the polymorphic CS calculus

Type compilation

$$\begin{array}{llllllllll}
\mathcal{V}[\![\alpha]\!] & = & \alpha & \mathcal{V}[\![base]\!] & = & base & \mathcal{V}[\![A \xrightarrow{Loc} B]\!] & = & Clo(\mathcal{V}[\![A]\!] \xrightarrow{Loc} C[\![B]\!]) \\
\mathcal{V}[\![\forall\alpha.A]\!] & = & \forall\alpha.C[\![A]\!] & \mathcal{V}[\![A \times B]\!] & = & \mathcal{V}[\![A]\!] \times \mathcal{V}[\![B]\!] & \mathcal{V}[\![\forall l.A]\!] & = & Clo(\forall l.C[\![A]\!]) \\
C[\![A]\!] & = & T\ (\mathcal{V}[\![A]\!]) & & & & & &
\end{array}$$

Term & value compilation

$$\begin{array}{lcl}
\mathcal{V}[\![x]\!]_{\Gamma,Loc,A} & = & x \\
\mathcal{V}[\![\lambda^{Loc'} x.M]\!]_{\Gamma,Loc,A \xrightarrow{Loc'} B} & = & clo(\overline{z}, F[\overline{l}, \overline{\alpha}]) \text{ where } \Gamma = \{\overline{l}, \overline{\alpha}, \overline{z} : \overline{C}\},\ F_{name} \text{ fresh}, \\
 & & (F_{name} : \overline{l}\ \overline{\alpha}.\overline{\mathcal{V}[\![C]\!]}.\mathcal{V}[\![A]\!] \xrightarrow{Loc'} C[\![B]\!] = \overline{l}\ \overline{\alpha}.\overline{z}.\lambda x.C[\![M]\!]_{\Gamma,x:A,Loc',B}) \in \Phi_{Loc'} \\
\mathcal{V}[\![\Lambda l.V]\!]_{\Gamma,Loc,\forall l.A} & = & clo(\overline{z}, F[\overline{l}, \overline{\alpha}]) \text{ where } \Gamma = \{\overline{l}, \overline{\alpha}, \overline{z} : \overline{C}\},\ F_{name} \text{ fresh}, \\
 & & (F_{name} : \overline{l}\ \overline{\alpha}.\overline{\mathcal{V}[\![C]\!]}.\forall l.C[\![A]\!] = \overline{l}\ \overline{\alpha}.\overline{z}.\Lambda l.C[\![V]\!]_{\Gamma,l,Loc,A}) \in \Phi_{\mathbf{c}}, \Phi_{\mathbf{s}} \\
\mathcal{V}[\![\Lambda\alpha.V]\!]_{\Gamma,Loc,\forall\alpha.A} & = & \Lambda\alpha.C[\![V]\!]_{\Gamma,\alpha,Loc,A} \\
\mathcal{V}[\![(V,W)]\!]_{\Gamma,Loc,A\times B} & = & (\mathcal{V}[\![V]\!]_{\Gamma,Loc,A}, \mathcal{V}[\![W]\!]_{\Gamma,Loc,B}) \\
C[\![V]\!]_{\Gamma,Loc,A} & = & \mathsf{unit}\ (\mathcal{V}[\![V]\!]_{\Gamma,Loc,A}) \\
C[\![L\ M]\!]_{\Gamma,Loc,B} & = & \mathsf{do}\ f \leftarrow C[\![L]\!]_{\Gamma,Loc,A \xrightarrow{Loc} B}\ \mathsf{in}\ \mathsf{do}\ x \leftarrow C[\![M]\!]_{\Gamma,Loc,A}\ \mathsf{in}\ f(x) \\
C[\![L\ M]\!]_{\Gamma,\mathbf{c},B} & = & \mathsf{do}\ f \leftarrow C[\![L]\!]_{\Gamma,\mathbf{c},A \xrightarrow{\mathbf{s}} B}\ \mathsf{in}\ \mathsf{do}\ x \leftarrow C[\![M]\!]_{\Gamma,\mathbf{c},A}\ \mathsf{in}\ \mathsf{req}(f,x) \\
C[\![L\ M]\!]_{\Gamma,\mathbf{s},B} & = & \mathsf{do}\ f \leftarrow C[\![L]\!]_{\Gamma,\mathbf{s},A \xrightarrow{\mathbf{c}} B}\ \mathsf{in}\ \mathsf{do}\ x \leftarrow C[\![M]\!]_{\Gamma,\mathbf{s},A}\ \mathsf{in}\ \mathsf{call}(f,x) \\
C[\![L\ M]\!]_{\Gamma,Loc,B} & = & \mathsf{do}\ f \leftarrow C[\![L]\!]_{\Gamma,Loc,A \xrightarrow{Loc'} B}\ \mathsf{in}\ \mathsf{do}\ x \leftarrow C[\![M]\!]_{\Gamma,Loc,A}\ \mathsf{in}\ \mathsf{gen}(Loc', f, x) \\
C[\![M[B]]\!]_{\Gamma,Loc,A\{B/\alpha\}} & = & \mathsf{do}\ f \leftarrow C[\![M]\!]_{\Gamma,Loc,\forall\alpha.A}\ \mathsf{in}\ f[\mathcal{V}[\![B]\!]] \\
C[\![M[Loc']]\!]_{\Gamma,Loc,A\{Loc'/l\}} & = & \mathsf{do}\ f \leftarrow C[\![M]\!]_{\Gamma,Loc,\forall l.A}\ \mathsf{in}\ f[Loc'] \\
C[\![(L,M)]\!]_{\Gamma,Loc,A\times B} & = & \mathsf{do}\ x \leftarrow C[\![L]\!]_{\Gamma,Loc,A}\ \mathsf{in}\ \mathsf{do}\ y \leftarrow C[\![M]\!]_{\Gamma,Loc,B}\ \mathsf{in}\ \mathsf{unit}\ (x,y) \\
C[\![\pi_i(M)]\!]_{\Gamma,Loc,A_i} & = & \mathsf{do}\ p \leftarrow C[\![M]\!]_{\Gamma,Loc,A_1\times A_2}\ \mathsf{in}\ \mathsf{let}\ x = \pi_i(p)\ \mathsf{in}\ \mathsf{unit}\ (x)
\end{array}$$

Figure 11: A typed compilation of $\lambda^{\forall}_{rpc}$ into $\lambda^{\forall}_{cs}$

$$\begin{array}{llll}
\text{Term} & m, n & ::= & v \mid \mathsf{let}\ x = m\ \mathsf{in}\ n \mid \pi_i(v) \mid v(w) \mid p(\overline{v}) \\
 & & \mid & \mathsf{case}\ e\ \mathsf{of}\ \overline{c\ \overline{x} \to m} \\
\text{Value} & v, w & ::= & x \mid (v,w) \mid Con\ \overline{v} \mid \mathsf{unit}\ v \mid \mathsf{do}\ x \leftarrow m\ \mathsf{in}\ n \\
\text{Primitive} & p & ::= & send \mid receive \\
\text{Prog.} & prg & ::= & (\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}}) \\
\text{Fun. map} & \Phi & ::= & \{\ F_1 = \overline{z}_1\lambda x_1.m_1,\ \cdots,\ F_n = \overline{z}_n\lambda x_n.m_n\ \}
\end{array}$$

Figure 12: The syntax for the untyped CS calculus

For implementation, we introduce an untyped language named λ_{cs} to be used as a target language for a type erasure translation retaining locations by value representation and exposing the concrete trampolined style communication. After presenting this translation, we will show that execution in λ_{cs} mirrors execution in $\lambda^{\forall}_{cs}$.

Figure 12 shows the syntax for an untyped CS calculus, which is a first-order functional programming language with networking. Terms denoted by m include *send* and *receive* as communication primitives. Case terms are included to deconstruct data constructor value, $Con\ \overline{v}$ where Con is a data constructor and $\overline{v}$ are its arguments. Note that values are denoted by v or w. For example, *Client* and *Server* are ordinary data constructors of type *Location* in λ_{cs} to represent location constants $\mathbf{c}$ and $\mathbf{s}$ in $\lambda^{\forall}_{cs}$, respectively. Another form of $Closure\ \overline{v}\ F_{name}$ is introduced to λ_{cs} to implement $clo(\overline{W}, F_{name}[\overline{Loc}\ \overline{\alpha}])$ in $\lambda^{\forall}_{cs}$ under the assumption

$$
\begin{array}{llll}
\text{Configuration} & conf & ::= & \langle m_{\mathbf{c}} \mid m_{\mathbf{s}} \rangle \\
\text{Eval. context} & e[\,] & ::= & e_{let}[\,] \mid \mathsf{do}\ x \leftarrow e[\,]\ \mathsf{in}\ m \\
 & e_{let}[\,] & ::= & [\,] \mid \mathsf{let}\ x = e_{let}[\,]\ \mathsf{in}\ m \\
\text{Conf. context} & \sigma & ::= & \langle e_1[\,] \mid e_2[\mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m] \rangle \\
 & & \mid & \langle e_1[\mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m] \mid e_2[\,] \rangle
\end{array}
$$

[Local reduction]

$$
\begin{array}{lll}
\text{(e-local)} & \dfrac{m \longrightarrow m'}{\sigma[m] \longrightarrow \sigma[m']} & \\
\text{(e-let)} & \mathsf{let}\ x = v\ \mathsf{in}\ m & \longrightarrow\ m\{v/x\} \\
\text{(e-do)} & \mathsf{do}\ x \leftarrow \mathsf{unit}\ v\ \mathsf{in}\ m & \longrightarrow\ m\{v/x\} \\
\text{(e-proj-i)} & \pi_i(v_1, v_2) & \longrightarrow\ v_i \quad \text{where } i = 1, 2 \\
\text{(e-app)} & (\mathit{Closure}\ \overline{w}\ f)(v) & \longrightarrow\ m_f\{\overline{w}/\overline{z}\}\{v/x\} \\
 & & \quad \text{if } \Phi(f) = \overline{z}.\lambda x.m_f \\
\text{(e-case)} & \mathsf{case}\ c_i\ \overline{v}\ \mathsf{of}\ \overline{c\ \overline{x} \rightarrow m} & \longrightarrow\ m_i\{\overline{v}/\overline{x}\}
\end{array}
$$

[Communication]

$$
\begin{array}{ll}
\text{(e-comm-c-s)} & \langle\ e_1[\mathsf{do}\ \mathit{send}\ v\ \mathsf{in}\ m_1] \mid e_2[\mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m_2]\ \rangle \\
 & \quad \longrightarrow\ \langle\ e_1[m_1] \mid e_2[m_2\{v/x\}]\ \rangle \\
\text{(e-comm-s-c)} & \langle\ e_1[\mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m_1] \mid e_2[\mathsf{do}\ \mathit{send}\ v\ \mathsf{in}\ m_2]\ \rangle \\
 & \quad \longrightarrow\ \langle\ e_1[m_1\{v/x\}] \mid e_2[m_2]\ \rangle
\end{array}
$$

Figure 13: The semantics for the untyped CS calculus

that $\overline{v}$ implements $\overline{Loc}$ together with $\overline{W}$ and F_{name} is $Con_{F_{name}}$. The payloads for remote procedure calls are represented by $\mathit{Apply}\ v\ w$ for $\mathsf{req}(V, W)$ and $\mathsf{call}(V, W)$ and by $\mathit{Ret}\ v$ for $\mathsf{unit}\ V$.

Function maps now hold codes with no free type and free location variables where some free term variables of the codes originate from the free location variables.

Configurations are in the form of $\langle m_{\mathbf{c}} \mid m_{\mathbf{s}} \rangle$ client term $m_{\mathbf{c}}$ and server term $m_{\mathbf{s}}$ as shown in Figure 13. Evaluation of an untyped CS program begins with $\langle \mathit{main} \mid \mathit{loop}_{body} \rangle$ where the main term is in the client side and loop_{body} is the body of trampoline loop. The definition of the trampoline loop is in Figure 14. A loop function at server waits for receiving payloads sent from client, serving them. Communication rules involve the two primitives, for example, as:

$$\langle \mathsf{do}\ \mathit{send}\ v\ \mathsf{in}\ m_1 \mid \mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m_2 \rangle \longrightarrow \langle m_1 \mid m_2\{v/x\} \rangle$$

for sending a value from the client to the server. Actually, each location has a trampoline loop for symmetric communication. The only difference is that $f(arg)$ in the Apply case of the loop function is implemented with one's own function store Φ_a.

The semantic rules are available in Figure 13. Evaluation contexts $e[\,]$ to choose a specific rule to execute are actually the same as the previous ones but configuration contexts $\sigma[\,]$ are notable. For example, client-side configuration contexts are in the form of $\langle e_1[\,] | e_2[\mathsf{do}\ x \leftarrow \mathit{receive}\ \mathsf{in}\ m] \rangle$ meaning that server is ready to receive any payloads that would be sent by the client during the evaluation of a term. When the term in the client is in the form of $\mathsf{do}\ \mathit{send}\ v\ \mathsf{in}\ m$, which is an abbreviation of $\mathsf{do}\ x \leftarrow \mathit{send}\ v\ \mathsf{in}\ m$ where x is unused, the client is about to send a payload v. Then the communication rule (e-comm-c-s) enables us to move the payload from the client to the server. For the opposite direction, server-side configuration contexts and (e-comm-s-c) will do that.

Function store compilation

$$
\begin{array}{lcl}
[\![\Phi_{\mathbf{c}}, \Phi_{\mathbf{s}}]\!] & = & \overline{[\![\Phi_{\mathbf{c}}]\!]_{\mathbf{c}}, [\![\Phi_{\mathbf{s}}]\!]_{\mathbf{s}}} \\
[\![\{\overline{F_{name} : Ty = Code}\}]\!]_a & = & \{\ \overline{F_{name} = [\![Code]\!]_a}\ \} \\
[\![\bar{l}\,\overline{\alpha}.\,\overline{z}.\mathit{OpenCode}]\!]_a & = & \overline{z_l}\,\overline{z}.[\![\mathit{OpenCode}]\!]_a \\
[\![\lambda x.M]\!]_a & = & \lambda x.[\![M]\!]_a \\
[\![\Lambda l.V]\!]_a & = & \lambda x_l.[\![V]\!]_a
\end{array}
$$

Location, term, and value compilation

$$
\begin{array}{l}
[\![l]\!] = x_l \qquad [\![\mathbf{c}]\!] = \mathit{Client} \qquad [\![\mathbf{s}]\!] = \mathit{Server} \\
[\![x]\!]_a = x \quad [\![(V, W)]\!]_a = ([\![V]\!]_a, [\![W]\!]_a) \quad [\![\pi_i(V)]\!]_a = \pi_i([\![V]\!]_a) \\
[\![\mathit{clo}(\overline{W}, F[\overline{Loc}][\overline{A}])]\!]_a = \mathit{Closure}\ (\overline{[\![Loc]\!]}\ \overline{[\![W]\!]_a})\ F \\
[\![\Lambda\alpha.\mathsf{unit}\ V]\!]_a = [\![V]\!]_a \quad [\![V[A]]\!]_a = \mathsf{unit}\ [\![V]\!]_a \\
[\![\mathsf{unit}\ V]\!]_a = \mathsf{unit}\ [\![V]\!]_a \\
[\![\mathsf{do}\ x \leftarrow M\ \mathsf{in}\ N]\!]_a = \mathsf{do}\ x \leftarrow [\![M]\!]_a\ \mathsf{in}\ [\![N]\!]_a \\
[\![\mathsf{req}(V, W)]\!]_a = \mathsf{do}\ \mathit{send}\ (\mathit{Apply}\ [\![V]\!]_a\ [\![W]\!]_a);\ \mathit{loop}\ () \\
[\![\mathsf{call}(V, W)]\!]_a = \mathsf{do}\ \mathit{send}\ (\mathit{Apply}\ [\![V]\!]_a\ [\![W]\!]_a);\ \mathit{loop}\ () \\
[\![\mathsf{gen}(\mathit{Loc}, V, W)]\!]_{\mathbf{c}} = \mathsf{if}([\![\mathit{Loc}]\!], [\![V(W)]\!]_{\mathbf{c}}, [\![\mathsf{req}(V, W)]\!]_{\mathbf{c}}) \\
[\![\mathsf{gen}(\mathit{Loc}, V, W)]\!]_{\mathbf{s}} = \mathsf{if}([\![\mathit{Loc}]\!], [\![\mathsf{call}(V, W)]\!]_{\mathbf{s}}, [\![V(W)]\!]_{\mathbf{s}}) \\
\quad \text{where } \mathsf{if}(v, m_1, m_2) = \mathsf{case}\ v\ \mathsf{of}\ \{\ \mathit{Client} \rightarrow m_1;\ \mathit{Server} \rightarrow m_2\ \} \\
[\![\mathsf{let}\ x = M\ \mathsf{in}\ N]\!]_a = \mathsf{let}\ x = [\![M]\!]_a\ \mathsf{in}\ [\![N]\!]_a \\
[\![V(W)]\!]_a = [\![V]\!]_a([\![W]\!]_a) \\
[\![V[\mathit{Loc}]]\!]_a = [\![V]\!]_a([\![\mathit{Loc}]\!]_a)
\end{array}
$$

Trampoline function

$$
\begin{array}{l}
\mathit{loop}\ u = \mathsf{do}\ x \leftarrow \mathit{receive} \\
\qquad \mathsf{case}\ x\ \mathsf{of} \\
\qquad\quad \mathit{Apply}\ f\ \mathit{arg} \rightarrow \mathsf{do}\ z \leftarrow f(\mathit{arg}) \\
\qquad\qquad\qquad\qquad\quad \mathit{send}\ (\mathit{Ret}\ z) \\
\qquad\qquad\qquad\qquad\quad \mathit{loop}\ () \\
\qquad\quad \mathit{Ret}\ y \rightarrow \mathsf{unit}\ y
\end{array}
$$

Figure 14: Compilation of $\lambda^{\forall}_{cs}$ into λ_{cs}

Local reduction rules are straightforward. A case reduction rule (e-case) is used to analyze location values in generic applications and to control the trampolined style communication flow. An application rule (e-app) extracts a function name from a closure, looks up Φ for its code, and continues to evaluate after substitutions on free variables and an argument.

Figure 14 shows compilation rules for locations, terms, values and function stores. Firstly, we review how to erase types and to compile locations. Every location variable l is replaced by a term variable x_l while the two location constants, $\mathbf{c}$ and $\mathbf{s}$, are compiled into Client and Server, respectively, that we explained previously.

Compiling a code, $\bar{l}\overline{\alpha}.\overline{z}.\mathit{OpenCode}$, erases the free type variables $\overline{\alpha}$, and changes the free location variables $\bar{l}$ into term variables. The compiled code will have $\overline{z_l}\,\overline{z}$ as free variables. Symmetrically, compiling a closure, $\mathit{clo}(\overline{W}, F_{name}[\overline{Loc}][\overline{A}])$, erases the free types $\overline{A}$, and lets the compiled closure hold $\overline{[\![Loc]\!]}\ \overline{[\![W]\!]}_a$ as values that come from translating the free locations $\overline{Loc}$ to variables $\overline{z_l}$ and from the existing values $\overline{W}$. Also, location applications are compiled essentially in the same way as term applications. Then the terms compiled from generic applications can examine locations by case term over values representing the locations.

Term applications $V(W)$ are compiled as an application term. Location applications $V[Loc]$ are compiled essentially in the same way but with the value representation $[\![Loc]\!]$ as an argument.

The compilation rules by definition guarantee a linear bound on the size of target terms too; in compiling $\mathsf{gen}(Loc, V, W)$, terms compiled from V and W can be hoisted out of the conditional.

The trampoline communication between the client and the server is supported by a key pattern do *send v* in *loop* () as used in compiling remote procedure call terms, $\mathsf{req}(V, W)$ and $\mathsf{call}(V, W)$. Here *loop* is a function waiting for receiving either *Apply f arg* to call $f(arg)$ locally and to return its result back to the other location, or *Ret y* to finish the trampoline communication. Both of $\mathsf{req}(V, W)$ and $\mathsf{call}(V, W)$ are compiled into a term in this pattern but at one's own location enforced by the $\lambda^{\forall}_{cs}$ type system. For $\mathsf{gen}(Loc, V, W)$, the compiled term has a case analysis on a value from the compiled location $[\![Loc]\!]$ to determine whether V is a remote procedure with an argument W.

For example, an untyped λ_{cs} program can be obtained from compiling the $\lambda^{\forall}_{cs}$ program in Section 4.1, as follows.

$$
\begin{array}{l}
\mathsf{do}\ \ h \leftarrow \mathsf{unit}\ \ (\mathit{Closure\ Server}\ f_2)\ \mathsf{in} \\
\mathsf{do}\ \{\ \mathit{send}\ (\mathit{Apply}\ h\ (\mathit{Closure}\ \emptyset\ f_3));\ \mathit{loop}\ ()\ \} \\
\text{where} \\
f_1 = \emptyset.\ \lambda x_1.\ \mathsf{unit}\ \ (\mathit{Closure}\ x_1\ f_2) \qquad \in \Phi_{\mathbf{c}}, \Phi_{\mathbf{s}} \\
f_2 = z_1.\ \lambda g.\ \mathsf{if}(\mathit{Client}, \mathsf{do}\ \{\ \mathit{send}\ (\mathit{Apply}\ g\ 1);\ \mathit{loop}\ ()\ \}, \ldots) \quad \in \Phi_{\mathbf{s}} \\
f_2 = z_1.\ \lambda g.\ \mathsf{if}(\mathit{Client}, \mathsf{case}\ g\ \mathsf{of}\ \mathit{Closure}\ \overline{w}\ f \rightarrow m, \ldots) \qquad \in \Phi_{\mathbf{c}} \\
\qquad \text{where } \Phi_{\mathbf{c}}(f) = \overline{z_f}.\lambda x_f.m_f, \quad m = m_f\{\overline{w}/\overline{z_f}\}\{1/x_f\} \\
f_3 = \emptyset.\ \lambda x.\ \mathsf{unit}\ x \qquad \in \Phi_{\mathbf{c}}
\end{array}
$$

Note that the code of f_2 in $\lambda^{\forall}_{cs}$ is compiled into two different λ_{cs} codes because $\mathsf{gen}(\mathbf{c}, g, 1)$ would be a remote call at server while it would be a local one at client.

Figure 15 shows a running of the untyped CS program example above. In the evaluation steps, note the following configuration

$$\langle\ e[(\mathit{Closure}\ \emptyset\ f_3)\ (1)]\ \mid\ e[\ \mathit{loop}_{body}]\ \rangle\ \ \text{in}\ \lambda_{cs}$$

where $e[\ ] = \mathsf{do}\ z \leftarrow [\ ]; \mathit{send}\ (\mathit{Ret}\ z);\ \mathit{loop}\ ()$. This λ_{cs} configuration actually mirrors a configuration $\langle\ \mathit{clo}(\emptyset, f_3)(1); [\ ]\ \mid\ [\ ]\ \rangle$ in $\lambda^{\forall}_{cs}$ that implements a term $(\lambda^{\mathbf{c}}x.x)\ 1$ at client intervening a remote call from client to server. This shows an example of how client and server trampoline loops in λ_{cs}, $e[\ ] \mid e[\mathit{loop}_{body}]$, implement client and server stacks in $\lambda^{\forall}_{cs}$, $[\ ] \mid [\ ]$.

Guided by the execution in $\lambda^{\forall}_{cs}$, our implementation with λ_{cs} is shown to respect the well-formed trampoline communication protocol by proving the semantic correctness of the compilation of $\lambda^{\forall}_{cs}$ into λ_{cs}. More details are found in the extended version [6].

Now we can prove the semantic correctness of the compilation of $\lambda^{\forall}_{cs}$ into λ_{cs}.

Theorem 5.1 (Semantic Correctness of Compilation of $\lambda^{\forall}_{cs}$ into λ_{cs}). *If* $\langle M|\epsilon\rangle \longrightarrow^* \langle \mathit{unit}\ V|\epsilon\rangle$ *in* $\lambda^{\forall}_{cs}$ *then* $\langle [\![M]\!]_{\mathbf{c}} \mid \mathit{loop}\ ()\rangle \longrightarrow^* \langle [\![\mathit{unit}\ V]\!]_{\mathbf{c}} \mid \mathit{loop}\ ()\rangle$ *in* λ_{cs}.

6 RELATED WORK AND DISCUSSION

Polymorphic locations: The polymorphic RPC calculus [5] was the first RPC calculus that supports polymorphic locations useful for writing succinct multi-tier programs. There are only a few publications that are relevant to the notion of polymorphic locations. ML5 has what they call *world polymorphism* based on modal logic, supporting *mobile code* runnable on different tiers represented by different possible worlds [16, 17]. The RPC calculi are not about the mobility of code.

Eliom [21–24] provides a macro feature called *shared sections*, which makes it possible to write code for the client and for the server at the same time, the third location called *base* such that codeat location base can be used both on the client and on the server, and *mixed* declarations from multiple locations in a single module. For the first and second features in Eliom, the polymorphic RPC calculus may serve as a theoretical foundation. Regarding the third feature, it would be interesting how the polymorphic RPC calculus can be extended with ML modules.

There are many questions left about programming with the RPC calculus. Are polymorphic locations useful? Judging from using similar features in the existing programming languages and our experience, this feature is useful for writing succinct programs. Is type-based control for remote procedure calls good? Polymorphic locations surely fit the type-based scheme. Although without any term-level distinctions, programmers could be confused, even with a term-level sign to signal remote procedure calls, such confusion would arise too. Rather the type-level information could help them to understand the RPC behavior. Is more than one location abstraction useful, and if so, for what? PolyRPC is still at an early stage, and so programming experiences with it are too limited to answer this question firmly. We could think of applying PolyRPC to more complex distributed programming, such as for the cloud [11] than the Web only with two locations. How can the PolyRPC compiler help programmers to avoid writing location annotations? For now, PolyRPC has a simple extension of bidirectional type checking [10] where programmers have to write all location applications, which can be burdensome sometimes. We could design some method to supply location arguments deduced from contexts as done in the context-aware programming languages [20].

Typed slicing compilations: The feature of slicing compilation is desirable in multi-tier programming languages because it can reduce code size at each location by stripping code that does not belong to the current location. More importantly, it can avoid unnecessary security leaks resulting from the server code being available at the client on the web browser where every detail of the (compiled JavaScript) code is exposed to reverse engineering.

Only a few multi-tier programming languages have supported slicing compilation . The untyped and monomorphic RPC calculi [4, 7] supported a slicing compilation but type information became unavailable after it while our slicing compilation produces typed polymorphic CS calculus programs. Links [7] has a slicing compilation method for the so called *stateless* server scheme but does not use it anymore. The source program is compiled into an intermediate representation tree, and the client portions of the tree are compiled to JavaScript and the server portions are directly interpreted. Ur/Web [2] supports a slicing compilation in implementation but there is no formal description of it. Eliom [21–24] has both theory and implementation of a typed slicing compilation generating OCaml programs. ScalaLoci [29, 31], Hop.js [28], and Gavial [25] do not separate the client part from the server part for running multi-tier programs. The multi-tier calculus [18, 19] is equipped with a typed slicing compilation used to optimize the

$$
\begin{array}{ll}
 & \langle\ main \mid loop_{body}\ \rangle \ \text{ where } loop_{body} = \text{do } x \leftarrow receive;\ \text{case } x \text{ of } \{\ Apply\ f\ arg \rightarrow \cdots\ ;\ Ret\ y \rightarrow \text{unit } y\ \} \\
= & \langle \text{do } h \leftarrow \text{unit } (Closure\ Server\ f_2) \text{ in } \text{ do } \{\ send\ (Apply\ h\ (Closure\ \emptyset\ f_3));\ loop\ ()\ \}\ \mid\ loop_{body}\ \rangle \\
\longrightarrow & \langle\ \text{do } \{\ send\ (Apply\ (Closure\ Server\ f_2)\ (Closure\ \emptyset\ f_3));\ loop\ ()\ \} \mid\ loop_{body}\ \rangle \\
\longrightarrow^2 & \langle\ loop_{body}\ \mid\ e[(Closure\ Server\ f_2)\ (Closure\ \emptyset\ f_3)]\ \rangle \ \text{ where } e[\ ] = \text{do } z \leftarrow [\ ]; send\ (Ret\ z);\ loop\ () \\
\longrightarrow & \langle\ loop_{body}\ \mid\ e[m_{f_2}\{Server/z_1\}\{(Closure\ \emptyset\ f_3)/g\}]\ \rangle \ \text{ where } m_{f_2} \text{ is the body of } f_2 \\
= & \langle\ loop_{body}\ \mid\ e[\text{do } \ send\ (Apply\ (Closure\ \emptyset\ f_3)\ 1);\ loop\ ()\ ]\ \rangle \\
\longrightarrow^2 & \langle\ e[(Closure\ \emptyset\ f_3)\ (1)]\ \mid\ e[\ loop_{body}]\ \rangle \\
\longrightarrow & \langle\ e[m_{f_3}\{1/x\}]\ \mid\ e[\ loop_{body}]\ \rangle \ \text{ where } m_{f_3} \text{ is the body of } f_3 \\
= & \langle\ e[\text{unit } 1]\ \mid\ e[\ loop_{body}]\ \rangle \\
\longrightarrow & \langle\ send\ (Ret\ 1);\ loop\ ()\ \mid\ e[\ loop_{body}]\ \rangle \\
\longrightarrow^2 & \langle\ loop_{body}\ \mid\ e[\ \text{unit } 1]\ \rangle \\
\longrightarrow & \langle\ loop_{body}\ \mid\ send\ (Ret\ 1);\ loop\ ()\ \rangle \\
\longrightarrow^2 & \langle\ \text{unit } 1\ \mid\ loop_{body}\ \rangle
\end{array}
$$

Figure 15: Evaluation steps for the running example of untyped λ_{cs} program

sliced code correctly. But their slicing compilation scheme is different from ours in that every sliced program from the scheme has the same control structure as the multitier program. ML5 [16, 17] had a formal description and implemented it with no correctness proofs.

Intensional location polymorphism and location representations: The idea of runtime location representations and checking in our CS calculi is closely connected with the existing runtime type analysis [1, 9, 14, 32]. For example, the use of generic applications on locations is analogous to the intensional polymorphism using `typecase` on types [14], and the use of the location representations is similar to the intentional polymorphism in type-erasure semantics [9]. A difference is that our study can guarantee the running of functions at the right place in the client-server model. With generalized algebraic data types (GADT) [1, 32], how to encode locations was discussed in [5], and is used in our implementation. But they did not design any type-erasure translation nor prove its correctness.

7 CONCLUSION AND FUTURE WORK

In this paper, we provided the first implementation of the polymorphic RPC calculus by the typed slicing compilation into the polymorphic CS calculus that is subsequently supported by the type-erasure, runtime location representation, and explicit communication primitives. The combination of static and dynamic resolution of local or remote procedure calls is new. This is different from the previous approaches only with dynamic resolution as in Links (the untyped RPC calculus) and only with static resolution as in the typed or polymorphic RPC calculus. We designed an experimental multi-tier programming language system for the Web, and developed a multi-tier ToDoMVC program as a case study.

As future work, we plan to enhance our bidirectional type checker for programmers to avoid having to write location parameters explicitly. An approach would automatically infer all location parameters while allowing programmers to write some only wherever necessary. Another direction is to expand the typed slicing compilation to the remaining untyped stage. An advanced calculus with GADTs and session types could serve as a target calculus for the purpose. As a benefit, conventional optimization methods would be made use of to optimize our dynamic approach, and the polymorphic RPC calculus could be implemented on top of the concurrent lambda calculus.

ACKNOWLEDGMENTS

We would like to thank Simon Fowler and the anonymous reviewers for helpful feedback and suggestions for improvement. This work was supported by ERC Consolidator Grant Skye (grant number ERC 682315), and by an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS). Kwanghoon Choi was supported by the National Research Foundation of Korea (NRF) grant funded by MoE (No. 2019R1I1A3A01058608). Sam Lindley was supported by the UKRI Future Leaders Fellowship EHOP (grant number MR/T043830/1).

REFERENCES

[1] James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report CUCIS TR2003-1901. Cornell University.

[2] Adam Chlipala. 2015. An Optimizing Compiler for a Purely Functional Web-Application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 10–21. https://doi.org/10.1145/2784731.2784741

[3] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. *SIGPLAN Not.* 50, 1 (Jan. 2015), 153–165. https://doi.org/10.1145/2775051.2677004

[4] Kwanghoon Choi and Byeong-Mo Chang. 2019. A theory of RPC calculi for client-server model. *Journal of Functional Programming* 29 (2019), e5. https://doi.org/10.1017/S0956796819000029

[5] Kwanghoon Choi, James Cheney, Simon Fowler, and Sam Lindley. 2020. A polymorphic RPC calculus. *Science of Computer Programming* 197 (2020), 102499. https://doi.org/10.1016/j.scico.2020.102499

[6] Kwanghoon Choi, James Cheney, Sam Lindley, and Bob Reynders. 2021. A Typed Slicing Compilation of the Polymorphic RPC Calculus. arXiv:2107.10793 [cs.PL]

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'06)*. Springer-Verlag, Berlin, Heidelberg, 266–296. http://dl.acm.org/citation.cfm?id=1777707.1777724

[8] Ezra Cooper and Philip Wadler. 2009. The RPC Calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/1599410.1599439

[9] Karl Crary, Stephanie Weirich, and Greg Morrisett. 2002. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming* 12, 6 (2002), 567–600. https://doi.org/10.1017/S0956796801004282

[10] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston,

Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582

[11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. *SIGPLAN Not.* 46, 12 (Sept. 2011), 118–129. https://doi.org/10.1145/2096148.2034690

[12] Simon Fowler. 2020. Model-View-Update-Communicate: Session Types Meet the Elm Architecture. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.14

[13] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. 1999. Trampolined Style. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming* (Paris, France) *(ICFP '99)*. ACM, New York, NY, USA, 18–27. https://doi.org/10.1145/317636.317779

[14] Robert Harper and Greg Morrisett. 1995. Compiling Polymorphism Using Intensional Type Analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, New York, NY, USA, 130–141. https://doi.org/10.1145/199448.199475

[15] Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

[16] Tom Murphy, VII. 2008. *Modal Types for Mobile Code*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. Advisor(s) Harper, Robert and Crary, Karl. AAI3314655.

[17] Tom Murphy, VII., Karl Crary, and Robert Harper. 2008. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing* (Sophia-Antipolis, France) *(TGC'07)*. Springer-Verlag, Berlin, Heidelberg, 108–123. http://dl.acm.org/citation.cfm?id=1793574.1793585

[18] Matthias Neubauer. 2007. *Multi-tier programming*. Ph.D. Dissertation. Universität Freiburg.

[19] Matthias Neubauer and Peter Thiemann. 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* (Long Beach, California, USA) *(POPL '05)*. ACM, New York, NY, USA, 221–232. https://doi.org/10.1145/1040305.1040324

[20] Tomas Petricek. 2017. *Context-aware programming languages*. Ph.D. Dissertation. University of Cambridge.

[21] Gabriel Radanne. 2017. *Tierless Web programming in ML*. Ph.D. Dissertation. University Paris Diderot.

[22] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: Tierless Web Programming from the Ground Up. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages* (Leuven, Belgium) *(IFL 2016)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/3064899.3064901

[23] Gabriel Radanne and Jérôme Vouillon. 2018. Tierless Web Programming in the Large. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 681–689. https://doi.org/10.1145/3184558.3185953

[24] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 377–397.

[25] Bob Reynders, Frank Piessens, and Dominique Devriese. 2020. Gavial: Programming the Web with Multi-tier FRP. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 1–32. https://doi.org/10.22152/programming-journal.org/2020/4/6

[26] Manuel Serrano and Gérard Berry. 2012. Multitier Programming in Hop. *Commun. ACM* 55, 8 (Aug. 2012), 53–59. https://doi.org/10.1145/2240236.2240253

[27] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a Language for Programming the Web 2.0. In *Proceedings of the 1st Dynamic Languages Symposium*. ACM, Portland, OR, USA, 975–985. https://doi.org/10.1145/1176617.1176756

[28] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. *SIGPLAN Not.* 51, 9 (Sept. 2016), 180–192. https://doi.org/10.1145/3022670.2951916

[29] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[30] Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:29. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3

[31] Pascal Weisenburger and Guido Salvaneschi. 2020. Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala. *The Art, Science, and Engineering of Programming* 4, 3, Article 17 (Feb. 2020), 29 pages.

[32] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. *SIGPLAN Not.* 38, 1 (Jan. 2003), 224–235. https://doi.org/10.1145/640128.604150