On Recovering Static Location Contexts in Links (A progress report)

Kwanghoon Choi

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

Abstract

This progress report describes an attempt to recover static location contexts in the Links-RPC calculus, which captures the RPC aspect of Links, by compiling to the polymorphic RPC calculus and back to itself. Currently, the proposed idea is limited to the rank-1 subset of the Links-RPC calculus. The limitation is caused by a technical problem involving polymorphic constructs in the adjustment of locations to resolve the location discrepancies resulting from our heuristic type compilation. We will continue to study to extend the applicability to the whole calculus by solving this problem.
Keywords: Location inference, Links, RPC calculus

1 Introduction

A motivation is this. The feature of location types would be good for multi-tier programming languages like Links in local computation optimization and static location inconsistency checking.

However, a naive integration of this feature in Links should be challenging because there might be a need to change several advanced features in a significant way. An alternative direction would pursue a way to recover static location contexts in Links equally useful for the aforementioned benefits but in the hope of no change or minimal changes that do not affect the Links language system so much.

Note that this paper refers to Links as its intermediate representation, the Links IR.

Figure 1 shows a general methodology to approach this alternative direction by analyzing static location contexts via compiling to the polymorphic RPC calculus and then by representing the analyzed context information in Links via compiling back.

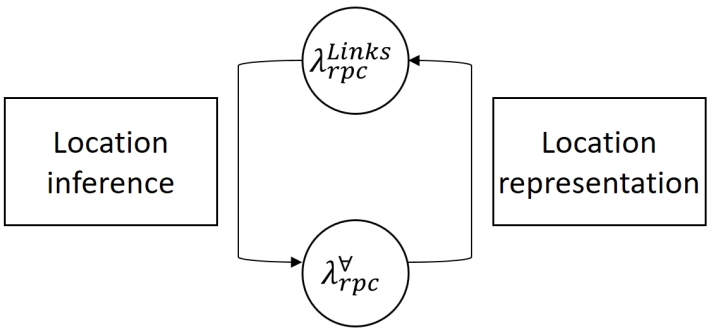

Figure 1: Recovering static location contexts by compiling to the polymorphic RPC calculus and back

Let us discuss location inference and representation by compiling to the polymorphic RPC calculus and subsequently compiling back. In Links, a client map function would be written as follows.

```
1  sig map : (a -> b, [a]) -> [b]
2  fun map(f, xs) client {
3    switch( xs ) {
4      case [] -> []
5      case y::ys -> f(y) :: map (f, ys)
6    }
7  }
```

As seen in the example, having a client location annotation enforces the map function to run at the client, but the location information is disregarded by types. Programmers may view the argument type as a client function type of $a \to b$ but the client map function could be applied to a server function of the same type. Therefore, the implementation of $f(y)$ should always check the location of the function f before its application to y.

A heuristic could be made up to reflect the programmers' view: by the presence of the client location annotation, all function types occurring in the map function types would be assumed to have the same location annotation. The following signature would describes the idea of this heuristic if Links did allow such a function type. For notation, $a \xrightarrow{client} b$ intends to mean a type of functions that are guaranteed to run in the client as in the polymorphic RPC calculus [2].

```
1  sig map : (a -client-> b, [a]) -client-> [b]
```

A client map function of this imaginary signature might not have to check the location of the function f in $f(y)$ anymore as long as f is guaranteed to be a client function as the signature. This is a local computation optimization.

When the map function is applied to a server function g of type $a \xrightarrow{server} b$, the location of g is required to be adjusted in some way to run this function inside the client map function that does not runtime checking. Choi and Chang [1] proposed an idea of using an eta conversion for the location adjustment as

$$fun\ (x)\ client\ \ g(x)$$

in the Links syntax[1]. This anonymous function is of type $a \xrightarrow{client} b$ as the signature. It would be responsible for moving to the server to perform the application of $g(x)$ while the map function is still running in the client without runtime location checking.

Thus our type inference strategy will consist of a heuristic of interpreting location annotation as explained and an adjustment of locations as the eta conversion in the example.

Generally, the map function is written as a location-neutral one where it should be able to run both of in the client and in the server. In Links, the map function is written without location annotation as follows.

```
1  sig map : (a -> b, [a]) -> [b]
2  fun map(f, xs) {
3    switch( xs ) {
4      case [] -> []
5      case y::ys -> f(y) :: map (f, ys)
6    }
7  }
```

Programmers could view this unannotated function as a polymorphic location function. This perception would be made use of when compiling into the polymorphic RPC calculus where programmers can write location abstractions and applications with location variables. Combining the previous heuristic with a location variable, a map function could be made up as a location polymorphic version where $map\ [client]$ is a client one and $map\ [server]$ is a server one.

```
1  sig map : forall l. (a -l-> b, [a]) -l-> [b]
```

Unfortunately, Links does not support the notion of location variables nor location abstractions and applications. So, we need to encode this location polymorphism in a way.

One way of encoding locations in Links would make use of GADTs as proposed in Choi et al [2]. Firstly, let $Location\ \alpha$ be a GADT whose data constructors are $Client$ and $Server$. They have type $Location\ ClientType$ and $Location\ ServerType$, respectively. The polymorphic map function of the signature would be written in Links, as follows.

```
1  sig map : Location l -> (a -> b, [a]) -> [b]
2  fun map(z) {
3    fun map'(f, xs) {
4      switch( xs ) {
5        case [] -> []
6        case y::ys -> f(y) :: map' (f, ys)
7    }
8  }
```

[1] Actually, Links does not allow to have location annotations in anonymous functions.

The Links signaure should be interpreted similarly as explained with the heuristic previously. That is, it would be viewed as $(a \xrightarrow{l} b, [a]) \xrightarrow{l} [b]$.

This polymorphic map function can run wherever it chooses to run by *map Client* or *map Server* by applying it to one of the data constructors that represent client and server locations. Interestingly, the polymorphic map function can still run as a local computation. So, there is no need to refer to z in the map example.

However, in other examples,there is sometimes a need to check locations in runtime. For this purpose, the polymorphic map function is written to take a value for representing locations as an argument. When runtime location checking is necessary, the argument z could be examined.

This is a glimpse of an idea of location representation when compiling back to Links.

Section 2 defines a Links-RPC calculus. Section 3 discusses a compilation method of $\lambda^{\forall}_{rpc}$ into λ^{Links}_{rpc}. Section 4 discusses a reverse direction compilation method. Section 5 concludes.

2 A Links-RPC Calculus

This section describes a RPC calculus that extends the RPC calculus [5] with unannotated λ-abstractions whose locations are not specified. The extended RPC calculus intends to capture the RPC feature of the Links programming language [4] in a simple way. For our setting, the calculus is also extended with polymorphic types with type abstractions and applications. Let us call it a Links-RPC calculus, λ^{Links}_{rpc}.

Syntax

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

Types

Type	A, B, C	$::=$	$base \mid A \to B \mid \alpha \mid A \times B \mid \forall\alpha.A$

Figure 2: The Links-RPC calculus λ^{Links}_{rpc}

The terms and types of the Links-RPC calculus are shown in Figure 2. In the calculus, it is not mandatory to annotate locations to λ abstractions. This is for programmers' convenience. An optional location $?a$ means that every annotation is either a location constant a or unknown. Regardless of the presence of location annotations, function types are $A \to B$ with no locations in types. No notion of location variables is available in the calculus, and so there are no location abstractions nor applications. Other than these differences, the terms and types are almost the same as those for the polymorphic RPC calculus, which is in the appendix for reference.

The semantics for λ^{Links}_{rpc} includes an evaluation rule for unannotated λ-abstractions, which once was proposed by the original RPC calculus [5], that evaluate to λ-abstractions annotated with the location of evaluation as

$$\text{(Unknown-Abs)}\ \frac{}{\lambda x.M \Downarrow_a \lambda^a x.M}$$

The typing rules are actually the same as for the System-F calculus except the appearance of annotated λ-abstractions whose locations are disregarded by types.

$$\text{(T-?a-Abs)}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda^{?a}x.M : A \to B}$$

For brevity, we omit writing the full definitions of the semantics and the typing rules for λ^{Links}_{rpc}.

3 Compiling λ^{Links}_{rpc} into $\lambda^{\forall}_{rpc}$

The purpose of compiling the Links-RPC calculus into the polymorphic RPC calculus is to recover location information during the evaluation of λ^{Links}_{rpc} terms and to express it by located types of $\lambda^{\forall}_{rpc}$ explicitly. Such statically recovered location information would be made use of later for transformations and optimizations, such as avoiding unnecessary location checking at runtime for solely local computation.

Basically, this compilation would translate normal function types $A_1 \to A_2$ into located function types $A_1' \xrightarrow{Loc} A_2'$ for some A_is. Then where do locations, such as Loc, come from? Generally, they are nowhere in the Links-RPC terms and types. Therefore, a heuristic should be introduced to recover location information lack in the terms and types.

A heuristic approach is to view λ_{rpc}^{Links} types A as $\lambda_{rpc}^{\forall}$ types B where

- the *skeleton* of B obtained from erasing all locations from it is the same as A, and
- all the erased locations are assumed to be the same as a given location Loc.

For example, the type of a client map function, $(Int \to Int) \to [Int] \to [Int]$, can be viewed as $(Int \xrightarrow{\mathbf{c}} Int) \xrightarrow{\mathbf{c}} [Int] \xrightarrow{\mathbf{c}} [Int]$. As is seen, not only the two function types on the spine get client locations but the argument function type is also annotated with it.

This idea can be formulated as a type translation $[\![A]\!]_{Loc} = B$:

$$\begin{array}{llllllll} [\![\alpha]\!]_{Loc} & = & \alpha & [\![A \to B]\!]_{Loc} & = & [\![A]\!]_{Loc} \xrightarrow{Loc} [\![B]\!]_{Loc} & & \\ [\![base]\!]_{Loc} & = & base & [\![A \times B]\!]_{Loc} & = & [\![A]\!]_{Loc} \times [\![B]\!]_{Loc} & [\![\forall\alpha.A]\!]_{Loc} & = & \forall\alpha.[\![A]\!]_{Loc} \end{array}$$

Figure 3: A type compilation of λ_{rpc}^{Links} into $\lambda_{rpc}^{\forall}$

Note that the explained client map function type in $\lambda_{rpc}^{\forall}$ is obtained from $[\![(Int \to Int) \to [Int] \to [Int]]\!]_{\mathbf{c}}$ by extending it over list types as $[\![[A]]\!]_{Loc} = [[\![A]\!]_{Loc}]$ in a straightforward way.

Obviously, this heurisitc is not always satisfactory. For example, *map f list* is legitimate in λ_{rpc}^{Links} even when *map* of the client map function type takes a function of type $Int \xrightarrow{\mathbf{c}} Int$ but f of type $Int \xrightarrow{\mathbf{s}} Int$ is given as an argument in $\lambda_{rpc}^{\forall}$. This is because λ_{rpc}^{Links} can deal with this discrepancy by runtime location checking. However, there should be some adjustments to locations in compiled types and terms.

A type-directed location adjustment is formulated as the following form of judgments:

$$M : A \Rightarrow M' : A'$$

From a term M of type A, a new adjusted term M' is derived under the guidance of another type A' having the same skeleton as but possibly locatational discrepancies with A. For example,

$$f : Int \xrightarrow{\mathbf{s}} Int \;\Rightarrow\; (\lambda^{\mathbf{c}}x.f\ x) : Int \xrightarrow{\mathbf{c}} Int$$

where given the three inputs, the adjusted client function $\lambda^{\mathbf{c}}x.f\ x$ is derived. Then the adjusted function could be fed into the client map function as its first argument in $\lambda_{rpc}^{\forall}$.

A type-directed location adjustment to terms is formulated as Figure 4. The rules are defined in terms of type structure, actually doing the eta conversion for function, pair, polymorphic, polymorphic location types of given terms. By the conversion, they will adjust all discrepant locations of two function types to produce new terms of the adjusted types.

$$\frac{}{M : A \Rightarrow M : A} \qquad \frac{x : C \Rightarrow N : A \quad M\ N : B \Rightarrow M' : D}{M : A \xrightarrow{Loc_1} B \Rightarrow \lambda^{Loc_2}x.M' : C \xrightarrow{Loc_2} D} \qquad \frac{\pi_i M : A_i \Rightarrow M_i' : A_i' \quad (i = 1, 2)}{M : A_1 \times A_2 \Rightarrow (M_1, M_2) : A_1' \times A_2'}$$

$$\frac{M[\alpha] : A \Rightarrow M' : B}{M : \forall\alpha.A \Rightarrow \Lambda\alpha.M' : \forall\alpha.B} \qquad \frac{M[l] : A \Rightarrow M' : B}{M : \forall l.A \Rightarrow \Lambda l.M' : \forall l.B}$$

Figure 4: A type-directed location adjustment to terms

Using the rules, one can derive the example adjustment judgment for given f, the server function type, and the client function, producing the adjusted term, $\lambda^{\mathbf{c}}x.f\ x$.

Based on the heuristic type compilation with the type-directed location adjustment, we can define the compilation rules of λ_{rpc}^{Links} into $\lambda_{rpc}^{\forall}$ as shown in Figure 5.

Basically, the compilation rules forms a transformation $[\![\mathcal{D}]\!]_{\Delta,Loc} = \mathcal{D}'$ of typing derivations $\mathcal{D}$ in λ_{rpc}^{Links} into those $\mathcal{D}'$ in $\lambda_{rpc}^{\forall}$. Additionally, as contextual information in $\lambda_{rpc}^{\forall}$, the compilation method is defined to take a type environment Δ and a location Loc. The type environment Δ provides how types for free variables in λ_{rpc}^{Links} are heuristically compiled depending on locational contexts. The location Loc informs where terms being compiled are.

$$
\begin{array}{lcl}
[\![\Gamma \vdash x : A]\!]_{\Delta,Loc} & = & \dfrac{x : B \in \Delta}{\Delta \vdash_{Loc} x : B} \\[2ex]
[\![\Gamma \vdash \lambda^a x.M : A \to B]\!]_{\Delta,Loc} & = & \text{let } \Delta, x : [\![A]\!]_a \vdash_a M' : D \;=\; [\![\Gamma, x : A \vdash M : B]\!]_{(\Delta, x:[\![A]\!]_a), a} \\
 & & \Delta \vdash_{Loc} \lambda^a x.M' : [\![A]\!]_a \xrightarrow{a} D \\[2ex]
[\![\Gamma \vdash \lambda x.M : A \to B]\!]_{\Delta,Loc} & = & \text{let } l \text{ be fresh} \\
 & & \text{let } \Delta, l, x : [\![A]\!]_l \vdash_l M' : D \;=\; [\![\Gamma, x : A \vdash M : B]\!]_{(\Delta, l, x:[\![A]\!]_l), l} \\
 & & \dfrac{\dfrac{\Delta, l \vdash_{Loc} \lambda^l x.M' : [\![A]\!]_l \xrightarrow{l} D}{\Delta \vdash_{Loc} \Lambda l.\lambda^l x.M' : \forall l.([\![A]\!]_l \xrightarrow{l} D)}}{\Delta \vdash_{Loc} (\Lambda l.\lambda^l x.M')[Loc] : ([\![A]\!]_l \xrightarrow{l} D)\{Loc/l\}} \\[2ex]
[\![\Gamma \vdash M\ N : B]\!]_{\Delta,Loc} & = & \text{let } \Delta \vdash_{Loc} M' : C \xrightarrow{Loc'} D \;=\; [\![\Gamma \vdash M : A \to B]\!]_{\Delta,Loc} \\
 & & \text{let } \Delta \vdash_{Loc} N'_0 : C_0 \;=\; [\![\Gamma \vdash N : A]\!]_{\Delta,Loc} \\
 & & \text{let } N'_0 : C_0 \Rightarrow N' : C \\
 & & \Delta \vdash_{Loc} M'\ N' : D \\[2ex]
[\![\Gamma \vdash (M_1, M_2) : A_1 \times A_2]\!]_{\Delta,Loc} & = & \text{let } \Delta \vdash_{Loc} M'_i : A'_i \;=\; [\![\Gamma \vdash M_i : A_i]\!]_{\Delta,Loc} \text{ for } i = 1, 2 \\
 & & \Delta \vdash_{Loc} (M'_1, M'_2) : (A'_1 \times A'_2) \\[2ex]
[\![\Gamma \vdash \pi_i M : A_i]\!]_{\Delta,Loc} & = & \text{let } \Delta \vdash_{Loc} M' : A'_1 \times A'_2 \;=\; [\![\Gamma \vdash M : A_1 \times A_2]\!]_{\Delta,Loc} \\
 & & \Delta \vdash_{Loc} \pi_i\ M' : A'_i \\[2ex]
[\![\Gamma \vdash \Lambda\alpha.V : \forall\alpha.A]\!]_{\Delta,Loc} & = & \text{let } \Delta, \alpha \vdash_{Loc} V' : A' \;=\; [\![\Gamma, \alpha \vdash V : A]\!]_{(\Delta,\alpha),Loc} \\
 & & \Delta \vdash_{Loc} \Lambda\alpha.V' : \forall\alpha.A' \\[2ex]
[\![\Gamma \vdash M\ [B] : A\{B/\alpha\}]\!]_{\Delta,Loc} & = & \text{let } \Delta \vdash_{Loc} M' : \forall\alpha.C \;=\; [\![\Gamma \vdash M : \forall\alpha.A]\!]_{\Delta,Loc} \\
 & & \Delta \vdash_{Loc} M'[\ [\![B]\!]_{Loc}\ ] : C\{[\![B]\!]_{Loc}/\alpha\}
\end{array}
$$

Figure 5: A compilation of λ^{Links}_{rpc} into $\lambda^{\forall}_{rpc}$

One of the most important compilation rules is for λ-applications $M\ N$. After M and N are compiled into M' and N'_0, the compiled argument term is adjusted to the argument type of the compiled functional term.

Unannotated λ-abstractions are compiled into location abstractions that are subsequently instantiated with the location of the λ-abstractions.

Note that compiling $\lambda^{?a}x.M$ of type $A \to B$ at a locational context Loc extends the current type environment with a type binding of x and a heuristically compiled argument type, that is, $[\![A]\!]_a$ or $[\![A]\!]_l$ for some fresh location variable.

3.1 Problem: Violating Syntactic Restriction on Polymorphism

We have a problem in the formulation of the compilation of λ^{Links}_{rpc} into $\lambda^{\forall}_{rpc}$. Recall that type and location abstractions are defined in the form as $\Lambda\alpha.V$ and $\Lambda l.V$ where the bodies are in the form of values. The compilation of arbitrary ranked λ^{Links}_{rpc} terms could produce $\lambda^{\forall}_{rpc}$ terms that violate this syntactic restriction. For those λ^{Links}_{rpc} terms of polymorphic types under the rank-1 polymorphism, often called the Hindley-Milner style polymorphism, the proposed compilation rules will always produce legitimate terms keeping the polymorphic restriction in syntax.

The cause of the problem is that the eta-conversion of type and location abstractions leads to type and location applications in their bodies. Let us explain the cause by example. It is legitimate to have an adjustment as

$$\Lambda\alpha.(\lambda^{\mathbf{c}}x.x, 42) : \forall\alpha.(\alpha \xrightarrow{\mathbf{c}} \alpha \times Int) \quad \Rightarrow \quad \Lambda\alpha.(\lambda^{\mathbf{s}}y.(\lambda^{\mathbf{c}}x.x)\ y, 42) : \forall\alpha.(\alpha \xrightarrow{\mathbf{s}} \alpha \times Int)$$

when the body of the type abstraction is known. Note that the body of the type abstraction is in the form of values. Otherwise, when it is difficult to know how the body of the type abstraction is structured, it seems only to have

$$f : \forall\alpha.(\alpha \xrightarrow{\mathbf{c}} \alpha \times Int) \quad \Rightarrow \quad \Lambda\alpha.(\lambda^{\mathbf{s}}y.(\pi_1\ f[\alpha])\ y,\ \pi_2\ f[\alpha]) : \forall\alpha.(\alpha \xrightarrow{\mathbf{s}} \alpha \times Int)$$

where the presence of $\pi_2\ f[\alpha]$ causes the body of the compiled type abstraction to be beyond the form of values.

4 Compiling $\lambda^{\forall}_{rpc}$ back to λ^{Links}_{rpc} extended with GADTs

Now we study how to compile $\lambda^{\forall}_{rpc}$ terms back to λ^{Links}_{rpc} terms. For this, the presence of the notion of location in $\lambda^{\forall}_{rpc}$ should be encoded in λ^{Links}_{rpc} terms. GADTs can be used to encode locations as was discussed in [2].

For the compilation, a GADT, $Location\ \alpha$, is introduced to λ^{Links}_{rpc} to have two data constructors, $Client$ and $Server$:

- $Client \quad : \quad Location\ ClientType$
- $Server \quad : \quad Location\ ServerType$

where $ClientType$ and $ServerType$ are some types whose inhabitants are of no interest.

Accordingly, λ^{Links}_{rpc} is assumed to be extended with case analysis terms on values. For brevity, we only consider a special case term for the location GADT as

$$\mathsf{case}\ L\ \mathsf{of}\ Client \to M;\ Server \to N.$$

For example, one can write $\Lambda\alpha.\lambda x.\mathsf{case}\ x\ \mathsf{of}\ Client \to M;\ Server \to N$ in the extended λ^{Links}_{rpc}, and the term can be of type $\forall\alpha.\ Location\ \alpha \to A$ for some type A. This term can be applied as $(\cdots)\ [ClientType]\ Client$ or as $(\cdots)\ [ServerType]\ Server$.

Using this way of encoding locations, the compilation of locations in $\lambda^{\forall}_{rpc}$ into types in λ^{Links}_{rpc} can be defined as Figure 6.

$$L[\![\mathbf{c}]\!] \ = \ ClientType \quad L[\![\mathbf{s}]\!] \ = \ ServerType \quad L[\![l]\!] \ = \ Location\ \alpha_l$$

Figure 6: A location compilation of $\lambda^{\forall}_{rpc}$ into types in λ^{Links}_{rpc}

For compiling back, a reverse of the type compilation that uses a heuristic to introduce locations to function types is required. This time the reverse type compilation would erase locations annotated to function types. Also, location abstractions $\forall l.A$, which is not expressible in λ^{Links}_{rpc}, would be replaced by a combination of type abstractions and function types $\forall\alpha_l.Location\ \alpha_l \to A$.

Figure 7 defines a reverse type compilation based on the idea explained.

$$\begin{array}{lll} T[\![\alpha]\!] \ = \ \alpha & T[\![base]\!] \ = \ base & T[\![A \xrightarrow{Loc} B]\!] \ = \ T[\![A]\!] \to T[\![B]\!] \\ T[\![\forall\alpha.A]\!] \ = \ \forall\alpha.T[\![A]\!] & T[\![\forall l.A]\!] \ = \ \forall\alpha_l.Location\ \alpha_l \to T[\![A]\!] & \end{array}$$

Figure 7: A type compilation of $\lambda^{\forall}_{rpc}$ into types in λ^{Links}_{rpc}

Figure 8 shows a term compilation of $\lambda^{\forall}_{rpc}$ into λ^{Links}_{rpc}. There are three things to note. Firstly, the definition uses location representation by values. The representation method is defined by the compilation of locations into terms $[\![Loc]\!] = V$. The client location $\mathbf{c}$ is represented by the data constructor $Client$ while $\mathbf{s}$ is so by $Server$. Note that the representation method is reminiscent of one used in the compilation of the polymorphic CS calculus into the untyped CS [3].

$[\![\mathbf{c}]\!] = Client \quad [\![\mathbf{s}]\!] = Server \quad [\![l]\!] = x_l$

$[\![x]\!]_{\Gamma,Loc,A} = x$

$[\![(M,N)]\!]_{\Gamma,Loc,A\times B} = ([\![M]\!]_{\Gamma,Loc,A} , [\![N]\!]_{\Gamma,Loc,B})$

$[\![\pi_i\ M]\!]_{\Gamma,Loc,A_i} = \pi_i\ ([\![M]\!]_{\Gamma,Loc,A_1\times A_2})$ (i=1,2)

$[\![\lambda^{Loc'}x.M]\!]_{\Gamma,Loc,A\xrightarrow{Loc'}B} = \lambda^{Ann[\![Loc']\!]}x.\ [\![M]\!]_{(\Gamma,x:T[\![A]\!]),Loc',T[\![B]\!]}$

$[\![M\ N]\!]_{\Gamma,Loc,B} = [\![M]\!]_{\Gamma,Loc,A\xrightarrow{Loc}B}\ [\![N]\!]_{\Gamma,Loc,A}$ (local procedure call)

$[\![M\ N]\!]_{\Gamma,\mathbf{c},B} = [\![M]\!]_{\Gamma,\mathbf{c},A\xrightarrow{\mathbf{s}}B}\ [\![N]\!]_{\Gamma,\mathbf{c},A}$ (remote procedure call)

$[\![M\ N]\!]_{\Gamma,\mathbf{s},B} = [\![M]\!]_{\Gamma,\mathbf{s},A\xrightarrow{\mathbf{c}}B}\ [\![N]\!]_{\Gamma,\mathbf{s},A}$ (remote procedure call)

$[\![M\ N]\!]_{\Gamma,Loc,B} = if([\![Loc]\!],\ if([\![Loc']\!], f\ arg, f\ arg),\ if([\![Loc']\!], f\ arg, f\ arg))$
where $f = [\![M]\!]_{\Gamma,Loc,A\xrightarrow{Loc'}B}$ and $arg = [\![N]\!]_{\Gamma,Loc,A}$.
$if(L,M,N) = \mathsf{case}\ M\ \mathsf{of}\ Client \to M; Server \to N.$
(local/remote procedure calls depending on conditionals)

$[\![\Lambda l.V]\!]_{\Gamma,Loc,\forall l.B} = \Lambda\alpha.\lambda[\![l]\!].[\![V]\!]_{\Gamma,Loc,B}$

$[\![M\ [Loc']]\!]_{\Gamma,Loc,B\{Loc'/l\}} = [\![M]\!]_{\Gamma,Loc,B\{Loc'/l\}}\ [\ L[\![Loc']\!]\ ]\ [\ [\![Loc']\!]\ ]$

$[\![\Lambda\alpha.V]\!]_{\Gamma,Loc,\forall l.B} = \Lambda\alpha.[\![V]\!]_{\Gamma,Loc,B}$

$[\![M\ [A]]\!]_{\Gamma,Loc,B\{A/\alpha\}} = [\![M]\!]_{\Gamma,Loc,\forall\alpha.B}\ [\ T[\![A]\!]\ ]$

$Ann[\![a]\!] = a \quad Ann[\![l]\!] = (unknown)$

Figure 8: A term compilation of $\lambda^{\forall}_{rpc}$ into terms in λ^{Links}_{rpc}

Secondly, location abstractions are represented actually by λ-abstractions guarded by type abstractions to introduce a type variable for a parameter of the type $Location\alpha$ of the λ-variable. For example,

$$\Lambda l.\lambda^{l} f.f\ 42$$

would be compiled to

$$\Lambda\alpha.\lambda^{\mathbf{c}} y_l.\lambda f.f\ 42$$

where the type of y_l is $Location\ \alpha$. Then an application term in $\lambda^{\forall}_{rpc}$

$$(\Lambda l.\lambda^{l} f.f\ 42)\ [\mathbf{c}]$$

would be a term in λ^{Links}_{rpc}

$$(\Lambda\alpha.\lambda^{\mathbf{c}} y_l.\lambda f.f\ 42)\ [ClientType]\ Client$$

where the type of y_l is $Location\ ClientType$. After applying it to the client location value $Client$, this location can be examined through the value bound to y_l in runtime.

Thirdly, there are four compilation rules for application terms in $\lambda^{\forall}_{rpc}$. The first three rules for application terms provides static location contexts. In the static location contexts, there is no need to check the location of functions to invoke by the runtime system for λ^{Links}_{rpc}, though our compilation rules have not expressed this by using different syntactic terms such as $V(W)$ for local procedure calls and $\mathsf{req}(V,W)$ and $\mathsf{call}(V,W)$ for remote procedure calls as in the Client-Server calculus [3]. The last rule for application terms is about dynamic location contexts where location information is provided by two variables (or one constant and one variable) $[\![Loc]\!]$ and $[\![Loc']\!]$ that represent the location of the application term and the location of its function, respectively.

Note that $Ann[\![Loc]\!]$ determines annotations over locations Loc. Location variables disappear after the compilation, but they remain as term variables as explained previously.

5 Conclusion

This progress report describes an attempt to recover static location contexts in the Links-RPC calculus, which represents the RPC aspect of Links, by compiling to the polymorphic RPC calculus and back to itself.

We have identified a technical problem involving polymorphic constructs in the adjustment of locations to resolve the location discrepancies resulting from our heuristic type compilation. We will continue to study to solve this problem.

References

[1] Kwanghoon Choi and Byeong-Mo Chang. A theory of RPC calculi for client-server model. *Journal of Functional Programming*, 29:e5, 2019.

[2] Kwanghoon Choi, James Cheney, Simon Fowler, and Sam Lindley. A polymorphic rpc calculus. *Science of Computer Programming*, 197:102499, 2020.

[3] Kwanghoon Choi, James Cheney, Sam Lindley, and Bob Reynders. A typed slicing compilation of the polymorphic rpc calculus. submitted, May 2021.

[4] Ezra K Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Ezra K. Cooper and Philip Wadler. The rpc calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 231–242, New York, NY, USA, 2009. ACM.

A The Polymorphic RPC Calculus

This section reminds the reader of the polymorphic RPC calculus [2]. 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.

A.1 The Syntax and the Semantics

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

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

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

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

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

Figure 9: The polymorphic RPC calculus $\lambda^{\forall}_{rpc}$

Figure 9 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, 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.

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 environment	Γ	::=	$\emptyset \mid \Gamma, x : A \mid \Gamma, \alpha \mid \Gamma, l$

Typing Rules

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

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

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

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

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

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

Figure 10: A type system for the polymorphic RPC calculus

A.2 The Type System

Figure 10 shows a type system for the polymorphic RPC calculus [2] 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 $\{l_1, \cdots, l_n, \alpha_1, \cdots, \alpha_k, x_1 : A_1, \cdots, x_m : A_m\}$. They are used to keep track of a set of free location, type, and value variables in the context of a given term.

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). When $Loc = Loc'$, one can statically decide that it is a local procedure call. Otherwise, Loc is different from Loc'. When both locations are constants as $Loc = a$ and $Loc' = b$, $L\ M$ is statically found to be a remote procedure call: if $a = \mathbf{c}$ and $b = \mathbf{s}$, it is to invoke a server function from the client, and if $a = \mathbf{s}$ and $b = \mathbf{c}$, it is to invoke a client function from the server. When at least one of them is a location variable, we cannot make a decision statically. (T-Labs) and (T-Lapp) are similar to the typing rules for type abstraction and type application. (T-Labs) checks if its bound location variable does not appear in the type environment and in the contextual location. (T-Lapp) substitutes Loc' for all occurrences of a location variable l on λ-abstractions in M.

The type soundness of the type system for the polymorphic RPC calculus, which was formulated as Theorem 1 and was proved by [2], 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 thogh both are in the same syntax of lambda applications.

Theorem 1 (Type soundness for $\lambda^{\forall}_{rpc}$ [2])**.** *For a closed term M, if $\emptyset \vdash_a M : A$ and $M \Downarrow_a V$, then $\emptyset \vdash_a V : A$.*