The RPC calculus

Ezra elias kilty Cooper
University of Edinburgh

Philip Wadler
University of Edinburgh

ABSTRACT

Several recent language designs have offered a unified language for programming a distributed system, with explicit notation of locations; we call these "location-aware" languages. These languages provide constructs allowing the programmer to control the location (the choice of host, for example) where a piece of code should run, which can be useful for security or performance reasons. On the other hand, a central mantra of WWW system engineering prescribes that web servers should be "stateless": that no "session state" should be maintained on behalf of individual clients—that is, no state that pertains to the particular point of the interaction at which a client program resides. Many implementations of location-aware languages are not at home on the web: they hold some kind of client-specific state on the server. We show how to implement a symmetrical location-aware language on top of a stateless server.

1. INTRODUCTION

Designing a web server requires thinking carefully about user state and how to manage it. Unlike a desktop application, which deals with one user at a time, or a traditional networked system, which may handle multiple simultaneous requests but in a more controlled environment, a modest web system can expect to deal with tens or hundreds of thousands of users in a day, each one can have multiple windows open on the site simultaneously—and these users can disappear at any time without notifying the server. This makes it infeasible for a web server to maintain state regarding a user's session. The mantra of web programming is: Get the state out!—get it out of the server and into the client. An efficient web server will respond to each request quickly and then forget about it even quicker.

Nonetheless, several recent high-level programming language designs [15, 17, 16] allow the programmer the illusion of a persistent environment encompassing both client and server; let us call these "location-aware languages." This allows the programmer to move control back and forth freely between client and server, using local resources on either side as necessary, but still expressing the program in one language. This paper shows how to implement a location-aware language in an environment with a *stateful client* and a *stateless server*.

The term *stateless server*, as it is applied by web engineers, means not that the server has strictly no storage, but rather that the server should not hold state pertaining to every thread of control on every one of its clients, which indeed may terminate without notifying the server. Typically, web servers do store persistent data—in a database, for example—but this data represents lasting, important information, rather than the ephemeral data supporting the immediate state of a particular client process. The challenge, then, is to support a location-aware language, where control state is maintained as transparently as in any programming language, even though the stateless-server substrate requires explicitly managing this state.

Our technique involves three essential steps: defunctionalization *à la* Reynolds, CPS translation, and a trampoline [10] for tunnelling server-to-client requests within server responses.

CPS translation and defunctionalization were used by Matthews et al. [13] for a similar end, that of writing web interactions in direct style, rather than CPS. We adapt these techniques, adding a trampoline to the toolbox, to support RPC calls in a location-aware language.

A version of this feature is built into the Links language [5]. In the current version, only calls to top-level functions can pass control between client and server. Here we show how to relax the top-level restriction. In particular, the current, limited version requires just a CPS translation and a trampoline; defunctionalization is needed in implementing nested remote-function definitions.

The only form of state modeled by this calculus is the control state, or call stack. This is a very pure view of state; but the techniques used here could be expanded to handle other forms, as discussed in Section 6. In fact, in the Links language, there is no other form of state, although its utility is recovered through the use of multiple concurrent processes, as championed by the Erlang community. In particular, mutable cells can be simulated in such a language by a process that holds data in its local variables.

This paper.

We present a simple higher-order λ-calculus enriched with location annotations, allowing the programmer to indicate the location where a fragment of code should execute. The

Syntax

constants	c		
variables	x		
locations	$a,\ b$	$::=$	$\mathsf{c} \mid \mathsf{s}$
terms	L, M, N	$::=$	$LM \mid V$
values	V, W	$::=$	$\lambda^a x.N \mid x \mid c$
evaluation contexts	E	$::=$	$[\ ] \mid VE \mid EN$

Semantics

$$\boxed{M \Downarrow_a V}$$

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

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

Figure 1: The RPC calculus, $\lambda_{\mathbf{rpc}}$.

semantics of this calculus indicates where each computation step is allowed to take place. This can be seen as a semantics of a language with Remote Procedure Call (RPC) features built in.

We then give a translation from this calculus to a *first-order* calculus that models an asymmetrical client–server environment, and show that the translation preserves the locative semantics of the source calculus.

2. THE RPC CALCULUS

The RPC calculus, λ_{rpc}, is defined in Figure 1. It is like an ordinary call-by-value calculus, but with location tags on λ-abstractions, denoting the location where the function body executes. We use the location set $\{\mathsf{c}, \mathsf{s}\}$ because we are interested in the client-server setting, but the calculus would be undisturbed by any other choice of location set. Constants are assumed to be universal, that is, all locations treat them the same way, and they contain no location annotations.

The semantics is defined by a big-step reduction relation $M \Downarrow_a V$, which is read, "the term M, evaluated at location a, results in value V." The reader can verify that the lexical body N of an a-annotated abstraction $\lambda^a x.N$ is only ever evaluated at location a, and thus the location annotations are honored by the semantics. During evaluation, however, that body may invoke other functions that evaluate at other locations. We write $N\{V/x\}$ for the capture-avoiding substitution of a value V for the variable x in the term N.

The semantics is equipped with a (partial) function δ which gives the action $\delta(a, c, W)$ at location a of a primitive function c on argument W. When δ is undefined, such an application is stuck. We do not require that the constants agree with themselves across locations—they might produce different values for the same arguments at different locations. This models the way, for example, fetching the time at different machines can give different values. In the calculus as it stands, however, such overloading is unobservable since every function application has a single execution location, dictated by the ^{a} on the nearest enclosing λ. (We also wish to study a variant where execution locations can be non-deterministic, but this is deferred for now.)

Example.

An example will illustrate the function of all these location annotations. For readability, this example avails of the syntactic sugar $\mathsf{let}^a\ x = M\ \mathsf{in}\ N$ for $(\lambda^a x.N)M$.

```
let^c getCredentials = λ^c prompt.
  let^c y = print prompt in
  read
in
let^c authenticate = λ^s x.
  let^s creds = getCredentials "Enter name, password:" in
  if (equal(creds, "ezra:opensesame")) then
    "the secret document"
  else "Access denied"
in authenticate()
```

Here let *print* and *read* be client-side functions, while *equal* is server-side Also imagine that we have extended the calculus with boolean constants and a conditional construct, $\mathsf{if}\ M\ \mathsf{then}\ N_t\ \mathsf{else}\ N_f$.

This example begins on the client and invokes the server-side function *authenticate*. This server-side function begins by calling the client function *getCredentials*, thus moving control back to the client, where *getCredentials* prints a prompt and reads back the credentials (username and password, say) from the client. The fact that *getCredentials* is defined by a λ^{c} abstraction means that its body will run at the client, and so the calls to *print* and *read* can take place locally, rather than with a move across the network for each call.

Furthermore, the fact that *authenticate* is defined by a λ^{s} abstraction means that its code should not be available to the client. Hence the constant `"ezra:opensesame"`, corresponding to the correct username and password, will not be revealed.

3. CLIENT/SERVER CALCULUS

Our target, λ_{cs}, defined in Figure 2, models a pair of interacting agents, a client and a server, each a first-order calculus.

Being first-order, the application form $f(\vec{M})$ is n-ary and allows only a function name, f, in the function position. The calculus also introduces constructor applications of the form $F(\vec{M})$, which can be seen as a tagged tuple. Constructor applications are destructed by the case-analysis form $\mathsf{case}\ M\ \mathsf{of}\ \mathcal{A}$. A list $\mathcal{A}$ of case alternatives is well-formed if it defines each name only once.

The client may make requests to the server, using the form $\mathsf{req}\ f\ (\vec{M})$. The server cannot make requests and can only run in response to client requests. Note that the $\mathsf{req}\ f\ (\vec{M})$ form has no meaning in server position; it may lead to a stuck configuration.

A configuration $\mathcal{K}$ of this calculus comes in one of two forms: a client-side configuration $(M; \cdot)$ consisting of an active client term M and a quiescent server (represented by the dot), or a server-side configuration $(E; M)$ consisting of an active server term M and a suspended client context E, which is waiting for the server's response. Although the client and server are in some sense independent agents, they interact in a synchronous fashion: upon making a request, the client blocks waiting for the server, and upon completing a request, the server is idle until the next request.

The construction of configurations visibly enforces the constraint that there be at most one request active between

Syntax

function names	f, g		
constructor names	F, G		
values	U, V, W	::=	$x \mid c \mid F(\vec{V})$
terms	L, M, N, X	::=	$x \mid c \mid f(\vec{M})$
			$\mid F(\vec{M}) \mid \mathsf{case}\ M\ \mathsf{of}\ \mathcal{A}$
			$\mid \mathsf{req}\ f\ (\vec{M})$
alternative sets	$\mathcal{A}$		a set of A items
case alternatives	A	::=	$F(\vec{x}) \Rightarrow M$
evaluation contexts	E	::=	$[\ ] \mid f(\vec{V}, E, \vec{M})$
			$\mid F(\vec{V}, E, \vec{M})$
			$\mid \mathsf{case}\ E\ \mathsf{of}\ \mathcal{A}$
			$\mid \mathsf{req}\ f\ (\vec{V}, E, \vec{M})$
configurations	$\mathcal{K}$	::=	$(M; \cdot) \mid (E; M)$
function definitions	D	::=	$f(\vec{x}) = M$
definition set	$\mathcal{D}, \mathcal{C}, \mathcal{S}$	::=	$\mathsf{letrec}\ D\ \mathsf{and}\ \cdots\ \mathsf{and}\ D$
continuation values	J, K	::=	$k \mid App(V, W, K) \mid F(\vec{V}, K)$

Semantics

$$\boxed{\mathcal{K} \longrightarrow_{\mathcal{C},\mathcal{S}} \mathcal{K}'}$$

Client:

$$(E[f(\vec{V})]; \cdot) \longrightarrow_{\mathcal{C},\mathcal{S}} (E[M\{\vec{V}/\vec{x}\}]; \cdot) \quad \text{if } (f(\vec{x}) = M) \in \mathcal{C}$$

$$(E[\mathsf{case}\ (F(\vec{V}))\ \mathsf{of}\ \mathcal{A}]; \cdot) \longrightarrow_{\mathcal{C},\mathcal{S}} (E[M\{\vec{V}/\vec{x}\}]; \cdot) \quad \text{if } (F(\vec{x}) \Rightarrow M) \in \mathcal{A}$$

Server:

$$(E; E'[f(\vec{V})]) \longrightarrow_{\mathcal{C},\mathcal{S}} (E; E'[M\{\vec{V}/\vec{x}\}]) \quad \text{if } (f(\vec{x}) = M) \in \mathcal{S}$$

$$(E; E'[\mathsf{case}\ (F(\vec{V}))\ \mathsf{of}\ \mathcal{A}]) \longrightarrow_{\mathcal{C},\mathcal{S}} (E; E'[M\{\vec{V}/\vec{x}\}]) \quad \text{if } (F(\vec{x}) \Rightarrow M) \in \mathcal{A}$$

Communication:

$$(E[\mathsf{req}\ f\ (\vec{V})]; \cdot) \longrightarrow_{\mathcal{C},\mathcal{S}} (E; f(\vec{V}))$$

$$(E; V) \longrightarrow_{\mathcal{C},\mathcal{S}} (E[V]; \cdot)$$

Figure 2: The client-server calculus ($\lambda_{\mathbf{cs}}$).

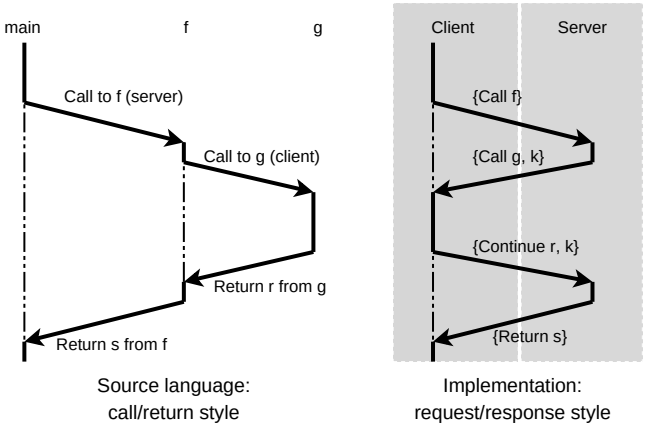

Figure 3: Simulation of $\lambda_{\mathbf{rpc}}$ calls by $\lambda_{\mathbf{cs}}$ requests.

this pair of client and server: there can be no back-and-forth call patterns, no nested calls.

Reduction takes place in the context of a pair of definition sets, one for each agent, thus the reduction judgment takes the form $\mathcal{K} \longrightarrow_{\mathcal{C},\mathcal{S}} \mathcal{K}'$. Each definition $f(\vec{x}) = M$ defines the function name f, taking arguments $\vec{x}$, to be the term M. The variables $\vec{x}$ are thus bound in M. A definition set is only well-formed if it uniquely defines each name. This does not preclude the other definition set, in a pair $(\mathcal{C}, \mathcal{S})$, from also defining the same name.

The reflexive, transitive closure of the relation $\longrightarrow_{\mathcal{C},\mathcal{S}}$ is written with a double-headed arrow $\twoheadrightarrow_{\mathcal{C},\mathcal{S}}$, where the definition-sets are fixed throughout the sequence.

3.1 Translation from λ_{rpc} to λ_{cs}

While the λ_{rpc} calculus allows arbitrarily deep nesting of control contexts between locations, λ_{cs} allows only for one frame of control to be waiting on the client while the server is running. To simulate arbitrarily nested control contexts, the translation uses the pattern seen in Figure 3. The left-hand diagram shows a sequence of calls between client and server functions. The solid line indicates the active line of control as it enters these calls, while the dashed line indicates a control context which is waiting for a function to return. In this example, *main* is a client function which calls a server function f which in turn calls a client function g.

The right-hand diagram shows the same series of calls as they occur at the low level. The dashed line again indicates a suspended control context. During the time when g has been invoked but has not yet returned a value, the server does not store the control context—or for that matter anything else about the ongoing computation—though at the high level f is still waiting on the server for the value from g. But the server's state is encapsulated in the value k, which it sends to the client along with a specification of the client-side call to perform, including a function reference and any arguments.

To orchestrate this interaction, the translation takes the single source program to two targets, one each for client and server. In so doing, each side's non-local functions are replaced by a *stub function*, which, rather than implementing the function directly, implements it by an RPC call to the other location. On the client side, this is easy: the client simply makes a request (with req) indicating the function

$$\boxed{(-)^\circ_- : V_{\lambda_{\mathrm{rpc}}} \to V_{\lambda_{\mathrm{cs}}}}$$

$$\begin{array}{rcll}
(\lambda^a x.N)^\circ & = & \ulcorner \lambda^a x.N \urcorner(\vec{y}) & \vec{y} = \mathrm{FV}(\lambda^a x.N) \\
x^\circ & = & x & \\
c^\circ & = & c &
\end{array}$$

$$\boxed{(-)^* : M_{\lambda_{\mathrm{rpc}}} \to M_{\lambda_{\mathrm{cs}}|\mathsf{c}}}$$

$$\begin{array}{rcll}
V^* & = & V^\circ & \\
(LM)^* & = & \mathit{apply}(L^*,\, M^*) &
\end{array}$$

$$\boxed{(-)^\dagger(-) : M_{\lambda_{\mathrm{rpc}}} \to V_{\lambda_{\mathrm{cs}}} \to M_{\lambda_{\mathrm{cs}}|\mathsf{s}}}$$

$$\begin{array}{rcll}
V^\dagger[\ ] & = & \mathit{cont}([\ ],\, V^\circ) & \\
(LM)^\dagger[\ ] & = & L^\dagger(\ulcorner M \urcorner(\vec{y}, [\ ])) & \text{where } \vec{y} = \mathrm{FV}(M)
\end{array}$$

Figure 4: Term-level translations from $\lambda_{\mathbf{rpc}}$ to $\lambda_{\mathbf{cs}}$.

$$\begin{array}{lcll}
\mathrm{coll}\ f\ (LM) & = & f(LM) \cup \mathrm{coll}\ f\ L \cup \mathrm{coll}\ f\ M & \\
\mathrm{coll}\ f\ (\lambda^a x.N) & = & f(\lambda^a x.N) \cup \mathrm{coll}\ f\ N & \\
\mathrm{coll}\ f\ V & = & f(V) & \text{if } V \neq \lambda^a x.N
\end{array}$$

Figure 5: Generic traversal function for $\lambda_{\mathbf{rpc}}$ terms.

and its arguments.

On the server side, the stub function *tunnels* its request through the response channel. It returns from its existing request context a representation of the call and the server-side continuation.

The client keeps a trampoline function on the stack underneath any server request, capable of recognizing the tunneled call and carrying it out. Upon completing this, it places a new request to the server, sending the call's result and the server continuation. The server resumes its computation by applying the continuation to the result.

Figures 4–7 give the translation. Figure 4 gives term-level translations $(-)^\circ$, $(-)^*$ and $(-)^\dagger$, which construct values, client terms, and server contexts, respectively. The $(-)^\dagger$ translation produces a context, which is expected to be filled by a continuation (which is a server value), so we will write $N^\dagger K$ for the translation of N to a server term with continuation K. The functions $(-)^*$ and $(-)^\dagger(-)$ are only defined for λ_{rpc} client and server terms, respectively. Functions $[\![-]\!]^{\mathsf{c},\mathrm{top}}$ (Fig. 6) and $[\![-]\!]^{\mathsf{s},\mathrm{top}}$ (Fig. 7) translate a source term to a definition set, making use of the generic traversal function coll (Fig. 5), which computes the union of the images under f of each subterm of a given term.

The function definitions for *apply*, *cont* and *tramp* are produced by the top-level translations, $[\![-]\!]^{\mathsf{c},\mathrm{top}}$ and $[\![-]\!]^{\mathsf{s},\mathrm{top}}$, defined in Figures 6–7. The former two, *apply* and *cont*, handle the defunctionalized dispatch to the appropriate function or continuation, while *tramp* is the trampoline, which tunnels server-to-client requests through responses. Let *arg* and k be special reserved variable names not appearing in the source program. The generic traversal function coll $f\ M$ of Figure 5 computes the union of the images under f of all the subterms M.

Rather than treating functions as if they already carry

$$\boxed{[\![-]\!]^{\mathsf{c},\mathrm{top}} : M_{\lambda_{\mathrm{rpc}}} \to \mathcal{D}_{\lambda_{\mathrm{cs}}}}$$

$$\begin{array}{rcl}
[\![M]\!]^{\mathsf{c},\mathrm{top}} & = & \mathsf{letrec}\ \mathit{apply}(\mathit{fun},\, \mathit{arg}) = \mathsf{case}\ \mathit{fun}\ \mathsf{of}\ [\![M]\!]^{\mathsf{c},\mathrm{fun}} \\
& & \mathsf{and}\ \mathit{tramp}(x) = \mathsf{case}\ x\ \mathsf{of} \\
& & \quad \mid \mathit{Call}(f,\, x,\, k) \Rightarrow \\
& & \qquad \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k,\, \mathit{apply}(f,\, x))) \\
& & \quad \mid \mathit{Return}(x) \Rightarrow x
\end{array}$$

$$\begin{array}{rcll}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\mathsf{c},\mathrm{fun},\mathrm{aux}} & = & \{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \Rightarrow N^*\{\mathit{arg}/x\}\} & \\
& & & \text{where } \vec{y} = \mathrm{FV}(\lambda x.N) \\
[\![\lambda^{\mathsf{s}} x.N]\!]^{\mathsf{c},\mathrm{fun},\mathrm{aux}} & = & & \\
\multicolumn{4}{l}{\{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \Rightarrow \mathit{tramp}(\mathsf{req}\ \mathit{apply}\ (\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}),\, \mathit{arg},\, \mathit{Fin}()))\}} \\
& & & \text{where } \vec{y} = \mathrm{FV}(\lambda x.N) \\
[\![M]\!]^{\mathsf{c},\mathrm{fun},\mathrm{aux}} & = & \{\} & \text{if } M \neq \lambda^a x.N \\
[\![M]\!]^{\mathsf{c},\mathrm{fun}} & = & \mathrm{coll}\ ([\![-]\!]^{\mathsf{c},\mathrm{fun},\mathrm{aux}})\ M &
\end{array}$$

Figure 6: Top-level translation, $\lambda_{\mathbf{rpc}}$ to $\lambda_{\mathbf{cs}}$ (client).

$$\boxed{[\![-]\!]^{\mathsf{s},\mathrm{top}} : M_{\lambda_{\mathrm{rpc}}} \to \mathcal{D}_{\lambda_{\mathrm{cs}}}}$$

$$\begin{array}{rcl}
[\![M]\!]^{\mathsf{s},\mathrm{top}} & = & \mathsf{letrec}\ \mathit{apply}(\mathit{fun},\, \mathit{arg},\, k) = \mathsf{case}\ \mathit{fun}\ \mathsf{of}\ [\![M]\!]^{\mathsf{s},\mathrm{fun}} \\
& & \mathsf{and}\ \mathit{cont}(k,\, \mathit{arg}) = \mathsf{case}\ k\ \mathsf{of} \\
& & \quad [\![M]\!]^{\mathsf{s},\mathrm{cont}} \\
& & \quad \mid \mathit{App}(\mathit{fun},\, k) \Rightarrow \mathit{apply}(\mathit{fun},\, \mathit{arg},\, k) \\
& & \quad \mid \mathit{Fin}() \Rightarrow \mathit{Return}(\mathit{arg})
\end{array}$$

$$\begin{array}{rcll}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\mathsf{s},\mathrm{fun},\mathrm{aux}} & = & & \\
\multicolumn{4}{l}{\{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \Rightarrow \mathit{Call}(\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}),\, \mathit{arg},\, k)\}} \\
& & & \text{where } \vec{y} = \mathrm{FV}(\lambda x.N) \\
[\![\lambda^{\mathsf{s}} x.N]\!]^{\mathsf{s},\mathrm{fun},\mathrm{aux}} & = & \{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \Rightarrow (N^\dagger k)\{\mathit{arg}/x\}\} & \\
& & & \text{where } \vec{y} = \mathrm{FV}(\lambda x.N) \\
[\![M]\!]^{\mathsf{s},\mathrm{fun},\mathrm{aux}} & = & \{\} & \text{if } M \neq \lambda^a x.N \\
[\![M]\!]^{\mathsf{s},\mathrm{fun}} & = & \mathrm{coll}\ ([\![-]\!]^{\mathsf{s},\mathrm{fun},\mathrm{aux}})\ M & \\
 & & & \\
[\![LM]\!]^{\mathsf{s},\mathrm{cont},\mathrm{aux}} & = & \{\ulcorner M \urcorner(\vec{y}, k) \Rightarrow M^\dagger(\mathit{App}(\mathit{arg}, k))\} & \\
& & & \text{where } \vec{y} = \mathrm{FV}(M) \\
[\![N]\!]^{\mathsf{s},\mathrm{cont},\mathrm{aux}} & = & \{\} & \text{if } N \neq LM \\
 & & & \\
[\![M]\!]^{\mathsf{s},\mathrm{cont}} & = & \mathrm{coll}\ ([\![-]\!]^{\mathsf{s},\mathrm{cont},\mathrm{aux}})\ M &
\end{array}$$

Figure 7: Top-level translation, $\lambda_{\mathbf{rpc}}$ to $\lambda_{\mathbf{cs}}$ (server).

unique labels (as in typical formalizations of defunctionalization), we assume an injective function that maps source terms into the space of constructor names. We write $\ulcorner M \urcorner$ for the name assigned to the term M by this function. An example is the function that collects the names assigned to immediate subterms and uses a hash function to digest these into a new name; issues of possible hash collisions would have to be treated delicately. We highlight the idea of using a function of the term itself because we wish for the labels to be robust in the face of server reboots—essential in the web environment—and even, perhaps, changes to unrelated parts of the code.

The bodies of the two *apply* functions will have a case for each abstraction appearing in the source term, regardless of location. For a location's own abstractions it gets a full definition but for the other's abstractions the case will be a mere stub. This stub dispatches a request to the other location, to apply the function to the given arguments.

The *cont* function is defined only on the server, because it arises from the CPS translation, which is only applied on the server side; it has a case for each continuation produced. This includes one for evaluating the argument subterm of each server-located application, one called *App* for applying a function to an argument, and one called *Fin* which returns a value to the client.

Recall the classic CPS translation for applications:

$$(LM)^{\text{cps}}K = L^{\text{cps}}(\underline{\lambda f.\, M^{\text{cps}}(\underline{\lambda x.\, fxK})}).$$

The outer underlined term corresponds to a continuation that is defunctionalized as $\ulcorner M \urcorner$. The inner one is defunctionalized as $App(f, K)$. Finally, recall that CPS always requires a "top-level" continuation, usually $\lambda x.x$, to extract a value from a CPS term; this corresponds to *Fin*.

The *tramp* function implements the trampoline. Its protocol is as follows: when the client first needs to make a server call, it makes a request wrapped in *tramp*. The server will either complete this call itself, without any client calls, or it will have to make a client call along the way. If it needs to make a client call, it returns a specification of that call as a value $Call(fun, arg, k)$, where *fun* and *arg* specify the call and k is the current continuation. The *tramp* function recognizes these constructions and evaluates the necessary terms locally, then places another request to the server to apply k to whatever value resulted, again wrapping the request in *tramp*. When the server finally completes its original call, it returns the value as the argument of the *Return* constructor; the *tramp* function recognizes this as the result of the original server call, so it simply returns x. As an invariant, *the client always wraps its server-requests in a call to tramp*. This way it can always handle *Call* responses.

To relate the two calculi, we use a reverse translation, from λ_{cs} to λ_{rpc}, given in Figure 8. All of the functions used in this translation are parameterized on the definition sets $\mathcal{C}$ and $\mathcal{S}$. The function $(-)^{\bullet}_{\mathcal{C},\mathcal{S}}$ takes λ_{cs} values to λ_{rpc} values. Next $(-)^{\star}_{\mathcal{C},\mathcal{S}}$ and $(-)^{\ddagger}_{\mathcal{C},\mathcal{S}}$ take client-side and server-side λ_{cs} terms (respectively) to λ_{rpc} terms. And $(-)^{\$}_{\mathcal{C},\mathcal{S}}$ takes λ_{cs} values representing continuations to λ_{rpc} evaluation contexts. The function $(-)^{\bullet}$ hits every value and $(-)^{\star}$ and $(-)^{\ddagger}(-)$ hit every λ_{rpc} term.

These functions are defined only on λ_{cs} terms and definitions in the range of the corresponding forward translation. In particular, $M^{\ddagger}_{\mathcal{C},\mathcal{S}}$ and $M^{\star}_{\mathcal{C},\mathcal{S}}$ are not defined unless *both*

$$\boxed{(-)^{\bullet}_{-,-} : V_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to V_{\lambda_{\text{rpc}}}}$$

$$\begin{aligned}
c^{\bullet}_{\mathcal{C},\mathcal{S}} &= c \\
x^{\bullet}_{\mathcal{C},\mathcal{S}} &= x \\
(F(\vec{V}))^{\bullet}_{\mathcal{C},\mathcal{S}} &= \lambda^{\mathsf{c}} x.(N\{x/arg\})^{\star}_{\mathcal{C},\mathcal{S}}\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} \\
&\quad \text{if } (F(\vec{y}) \Rightarrow N) \in \text{cases}(apply, \mathcal{C}) \text{ and } N \neq tramp(\mathsf{req}\ \cdot\ \cdot) \\
&\quad \text{where } x \text{ fresh for } \vec{y}, \vec{V} \\
(F(\vec{V}))^{\bullet}_{\mathcal{C},\mathcal{S}} &= \lambda^{\mathsf{s}} x.(N\{x/arg\})^{\ddagger}_{\mathcal{C},\mathcal{S}}\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} \\
&\quad \text{if } (F(\vec{y}) \Rightarrow N) \in \text{cases}(apply, \mathcal{S}) \text{ and } N \neq Call(\cdot,\cdot,\cdot) \\
&\quad \text{where } x \text{ fresh for } \vec{y}, \vec{V}
\end{aligned}$$

$$\boxed{(-)^{\star}_{-,-} : M_{\lambda_{\text{cs}}|\mathsf{c}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to M_{\lambda_{\text{rpc}}}}$$

$$\begin{aligned}
V^{\star}_{\mathcal{C},\mathcal{S}} &= V^{\bullet}_{\mathcal{C},\mathcal{S}} \\
(apply(L, M))^{\star}_{\mathcal{C},\mathcal{S}} &= L^{\star}_{\mathcal{C},\mathcal{S}} M^{\star}_{\mathcal{C},\mathcal{S}}
\end{aligned}$$

$$\boxed{(-)^{\ddagger}_{-,-} : M_{\lambda_{\text{cs}}|\mathsf{s}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to M_{\lambda_{\text{rpc}}}}$$

$$\begin{aligned}
(cont(K, V))^{\ddagger}_{\mathcal{C},\mathcal{S}} &= K^{\$}_{\mathcal{C},\mathcal{S}}[V^{\bullet}_{\mathcal{C},\mathcal{S}}] \\
(apply(V, W, K))^{\ddagger}_{\mathcal{C},\mathcal{S}} &= K^{\$}_{\mathcal{C},\mathcal{S}}[V^{\bullet}_{\mathcal{C},\mathcal{S}} W^{\bullet}_{\mathcal{C},\mathcal{S}}]
\end{aligned}$$

$$\boxed{(-)^{\$}_{-,-} : V_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to \mathcal{D}_{\lambda_{\text{cs}}} \to (M_{\lambda_{\text{rpc}}} \to M_{\lambda_{\text{rpc}}})}$$

$$\begin{aligned}
k^{\$}_{\mathcal{C},\mathcal{S}}[\ ] &= [\ ] \\
(App(V, K))^{\$}_{\mathcal{C},\mathcal{S}}[\ ] &= K^{\$}_{\mathcal{C},\mathcal{S}}[V^{\bullet}_{\mathcal{C},\mathcal{S}}[\ ]] \\
(F(\vec{V}, K))^{\$}_{\mathcal{C},\mathcal{S}}[\ ] &= K^{\$}_{\mathcal{C},\mathcal{S}}[[\ ](M\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\})] \\
&\quad \text{if } (F(\vec{y}, k) \Rightarrow M^{\dagger}(App(arg, k))) \in \text{cases}(cont, \mathcal{S}) \\
&\quad \text{and } F \neq Fin,\ F \neq App
\end{aligned}$$

Figure 8: Reverse translation, λ_{cs} to λ_{rpc}.

definition sets $\mathcal{C}$ and $\mathcal{S}$ define all the constructors appearing in M.

Provided $(\mathcal{C}, \mathcal{S})$ are in the image of $([\![-]\!]^{\text{c,top}}, [\![-]\!]^{\text{s,top}})$, then $K^{\$}_{\mathcal{C},\mathcal{S}}$ is a term context for λ_{rpc}; by a simple inductive argument we can see that it is always an evaluation context: it meets the grammar $E ::= [\ ] \mid VE \mid EM$.

To extract the alternatives of case-expressions from function bodies, we use a function cases, defined as follows:

Definition 1. *The function* cases *is defined by the rule:*

$$\frac{(f(x, \vec{y}) = \mathsf{case}\ x\ \mathsf{of}\ \mathcal{A}) \in \mathcal{D}}{\text{cases}(f, \mathcal{D}) = \mathcal{A}}$$

This relies on the fact that each of our special functions dispatches on the first of its arguments, whether that be the argument *fun* for *apply*, or k for *cont*; the dispatching argument is conveniently always the first.

3.2 Correctness

During reduction we may lose subterms which would have given rise to defunctionalized definitions; thus the reduction of a term does not have the same definition-set as its ancestor. Still, all the definitions it needs were generated by the original term; we formalize this as follows. The containment holds just when the names defined in the right-hand side are all defined in the left-hand side and upon inspecting corresponding function definitions, either the bodies are identical or they are both case analyses where the left-hand side contains all the alternatives of the right-hand side.

DEFINITION 2 (DEFINITION CONTAINMENT). *We say a definition set $\mathcal{D}$* contains *$\mathcal{D}'$, written $\mathcal{D} \geqslant \mathcal{D}'$, iff for each definition $f(\vec{x}) = M'$ in $\mathcal{D}'$ there is a definition $f(\vec{x}) = M$ in $\mathcal{D}$ and either $M = M'$ or* $\mathrm{cases}(f, \mathcal{D}) \supseteq \mathrm{cases}(f, \mathcal{D}')$.

LEMMA 1. *For any subterm M' of M, if $\mathcal{C} \geqslant [\![M]\!]^{\mathsf{c},\mathrm{top}}$ then $\mathcal{C} \geqslant [\![M']\!]^{\mathsf{c},\mathrm{top}}$ and if $\mathcal{S} \geqslant [\![M]\!]^{\mathsf{s},\mathrm{top}}$ then $\mathcal{S} \geqslant [\![M']\!]^{\mathsf{s},\mathrm{top}}$.*

Definitions produced by the top-level translations are *closed*: for each term that we find translated on the right-hand side of a definition case, all of that term's definitions can also be found amongst the definitions. More precisely:

LEMMA 2 (CLOSURE, DEFINITION SETS). *Let $\mathcal{C}$ and $\mathcal{S}$ be in the range of $[\![-]\!]^{\mathsf{c},\mathrm{top}}$ and $[\![-]\!]^{\mathsf{s},\mathrm{top}}$ respectively.*

- *If $N^*\{arg/x\}$ is the right-hand side of an element of* $\mathrm{cases}(apply, \mathcal{C})$ *then $\mathcal{C} \geqslant [\![N]\!]^{\mathsf{c},\mathrm{top}}$.*
- *If $(N^\dagger k)\{arg/x\}$ is the right-hand side of an element of* $\mathrm{cases}(apply, \mathcal{S})$ *then $\mathcal{S} \geqslant [\![N]\!]^{\mathsf{s},\mathrm{top}}$.*
- *If $M^\dagger(App(arg, k))$ is the right-hand side of an element of* $\mathrm{cases}(cont, \mathcal{S})$ *then $\mathcal{S} \geqslant [\![M]\!]^{\mathsf{s},\mathrm{top}}$.*

PROOF. Let X be the term such that $(\mathcal{C}, \mathcal{S}) = ([\![X]\!]^{\mathsf{c},\mathrm{top}}, [\![X]\!]^{\mathsf{s},\mathrm{top}})$. Each element of $\mathrm{cases}(apply, \mathcal{C})$ that has a right-hand side of the form $N^*\{arg/x\}$ is produced by a term $\lambda^{\mathsf{c}}x.N$, a subterm of X. As N is a subterm of X, then, $\mathcal{C} \geqslant [\![N]\!]^{\mathsf{c},\mathrm{top}}$. A similar argument holds for the other two consequents. □

LEMMA 3 (RETRACTION). *When $\mathcal{C} \geqslant [\![M]\!]^{\mathsf{c},\mathrm{top}}$ and $\mathcal{S} \geqslant [\![M]\!]^{\mathsf{s},\mathrm{top}}$, we have for each M*

(i) $(M^\circ)^\bullet_{\mathcal{C},\mathcal{S}} = M$ *provided M is a value,*

(ii) $(M^*)^\star_{\mathcal{C},\mathcal{S}} = M$, *and*

(iii) $(M^\dagger K)^\ddagger_{\mathcal{C},\mathcal{S}} = K^{\$}_{\mathcal{C},\mathcal{S}} M$ *for each K in λ_{cs}.*

PROOF. By induction on the structure of M. □

The reverse translation commutes with substitution.

LEMMA 4 (SUBSTITUTION). *Given definition sets $\mathcal{C}$, $\mathcal{S}$*

$$V^\bullet_{\mathcal{C},\mathcal{S}}\{W^\bullet_{\mathcal{C},\mathcal{S}}/x\} = (V\{W/x\})^\bullet_{\mathcal{C},\mathcal{S}}$$
$$M^\ddagger_{\mathcal{C},\mathcal{S}}\{W^\bullet_{\mathcal{C},\mathcal{S}}/x\} = (M\{W/x\})^\ddagger_{\mathcal{C},\mathcal{S}} \text{ and}$$
$$M^\star_{\mathcal{C},\mathcal{S}}\{W^\bullet_{\mathcal{C},\mathcal{S}}/x\} = (M\{W/x\})^\star_{\mathcal{C},\mathcal{S}}.$$

PROOF. By induction on V or M as appropriate. □

Now we show the soundness result, which is fairly routine.

LEMMA 5 (SOUNDNESS). *For any term M and substitution σ in λ_{rpc}, together with definition sets $\mathcal{C}$ and $\mathcal{S}$ such that $\mathcal{C} \geqslant [\![M]\!]^{\mathsf{c},\mathrm{top}}$ and $\mathcal{S} \geqslant [\![M]\!]^{\mathsf{s},\mathrm{top}}$, we have the following for all V and K:*

(i) $M^*\sigma \longrightarrow\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} V$ *implies* $M\sigma^\bullet_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{c}} V^\bullet_{\mathcal{C},\mathcal{S}}$ *and*

(ii) $tramp([\ ]); (M^\dagger K)\sigma \longrightarrow\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} tramp([\ ]); cont(K, V)$ *implies* $M\sigma^\bullet_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{s}} V^\bullet_{\mathcal{C},\mathcal{S}}$.

PROOF. By induction on the length of the λ_{cs} reduction sequence. Throughout the induction, σ is kept general.

When using the inductive hypothesis, the preconditions that $\mathcal{C} \geqslant [\![M]\!]^{\mathsf{c},\mathrm{top}}$ and $\mathcal{S} \geqslant [\![M]\!]^{\mathsf{s},\mathrm{top}}$ will be maintained because we will only use the inductive hypothesis on subterms of the M and on terms N whose translations are part of the rhs of definitions in $\mathcal{C}, \mathcal{S}$ and thus for which $\mathcal{C}, \mathcal{S} \geqslant [\![N]\!]^{\mathsf{c},\mathrm{top}}$ and $[\![N]\!]^{\mathsf{s},\mathrm{top}}$ (by the closure of definition-sets).

In this proof we omit the subscripts $\mathcal{C}$, $\mathcal{S}$ on reductions, because they are unchanged throughout reduction sequences, and on the reverse-translation functions, because they are unchanged throughout the recursive calls thereof.

Take cases on the structure of the starting term, either (i) $M^*\sigma$ or (ii) $(M^\dagger K)\sigma$, and split the conclusion into cases for (i) and (ii);

- Case LM for (i).

  By hypothesis, we have $(LM)^*\sigma \longrightarrow\!\!\!\!\rightarrow V$. By definition, $(LM)^*\sigma = apply(L^*\sigma, M^*\sigma)$.

  In order for the reduction not to get stuck, it must be that

  - $L^*\sigma \longrightarrow\!\!\!\!\rightarrow F(\vec{V})$ with $(F(\vec{V}))^\bullet = \lambda^a x.N\{\vec{V}^\bullet/\vec{y}\}$ for fresh x and some a and N.
  - $M^*\sigma$ reduces to a value; call it W.

  The freshness of x will allow us to equate simultaneous and sequential substitutions involving x.

  The reduction begins as follows:

  $$\begin{array}{rcl}(LM)^*\sigma & = & apply(L^*\sigma, M^*\sigma)\\ & \longrightarrow\!\!\!\!\rightarrow & apply(F(\vec{V}), M^*\sigma)\\ & \longrightarrow\!\!\!\!\rightarrow & apply(F(\vec{V}), W)\end{array}$$

  Applying the inductive hypothesis twice, using Lemma 2 to obtain the containment condition, we get

  $$L\sigma^\bullet \Downarrow_{\mathsf{c}} (F(\vec{V}))^\bullet = \lambda^{\mathsf{c}}x.N\{\vec{V}^\bullet/\vec{y}\},$$
  $$M\sigma^\bullet \Downarrow_{\mathsf{c}} W^\bullet \quad \text{and}$$

  We now show, by cases on a, the third leg of BETA, namely $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_a V^\bullet$.

  If $a = \mathsf{c}$ then N is such that $(F(\vec{y}) \Rightarrow N^*\{arg/x\}) \in \mathrm{cases}(apply, \mathcal{C})$, by definition of $(-)^\bullet$.

  Now the reduction finishes as:

  $$\begin{array}{rcl} & & apply(F(\vec{V}), W)\\ & \longrightarrow\!\!\!\!\rightarrow & N^*\{\vec{V}/\vec{y}, W/x\} = N^*\{\vec{V}/\vec{y}\}\{W/x\}\\ & \longrightarrow\!\!\!\!\rightarrow & V\end{array}$$

  So by the inductive hypothesis (invoking Lemma 2),

  $$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{c}} V^\bullet.$$

If $a = \mathsf{s}$ then N is such that

$(F(\vec{y}) \Rightarrow (N^\dagger k)\{arg/x\})) \quad \in \text{cases}(apply, \mathcal{S})$ and
$(F(\vec{y}) \Rightarrow tramp(\mathsf{req}\ apply\ (F(\vec{y}), arg, Fin()))) \quad \in \text{cases}(apply, \mathcal{C})$.

Now the reduction finishes as

$$\begin{array}{rl} & apply(F(\vec{V}), W) \\ \longrightarrow\!\!\!\!\rightarrow & tramp(\mathsf{req}\ apply\ (F(\vec{V}), W, Fin())) \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ apply(F(\vec{V}), W, Fin()) \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ (N^\dagger(Fin()))\{\vec{V}/\vec{y}, W/x\} \\ = & (N^\dagger(Fin()))\{\vec{V}/\vec{y}\}\{W/x\} \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ cont(Fin(), V) \\ \longrightarrow\!\!\!\!\rightarrow & V \end{array}$$

So by the inductive hypothesis (invoking Lemma 2),

$$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{c}} V^\bullet.$$

The judgment $(LM)\sigma^\bullet \Downarrow_{\mathsf{c}} V^\bullet$ follows by Beta. *huzzah!*

- Case LM for (ii).

  By hypothesis, we have for some K

  $$tramp([\ ]);\ ((LM)^\dagger K)\sigma \longrightarrow\!\!\!\!\rightarrow tramp([\ ]);\ cont(K, V).$$

  By definition, $(LM)^\dagger K = L^\dagger(\ulcorner M \urcorner(\vec{z}, K))$, letting $\vec{z} = \text{FV}(M)$.

  In order for the reduction not to get stuck, it must be that:

  1. $\ulcorner M \urcorner(\vec{z}, k) \Rightarrow M^\dagger(App(arg, k))$ is in cases$(cont, \mathcal{S})$
  2. $(M^\dagger K)\sigma$ reduces to a term of the form $cont(K, W)$ and
  3. $(L^\dagger K)\sigma$ reduces to a term of the form $cont(K, F(\vec{y}))$, with $(F(\vec{y}))^\bullet = \lambda^a x.N$ for fresh x and some a and N.

  The reduction begins as follows:

  $$\begin{array}{rl} & tramp([\ ]);\ ((LM)^\dagger K)\sigma \\ = & tramp([\ ]);\ (L^\dagger(\ulcorner M \urcorner(\vec{z}, K)))\sigma \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ cont((\ulcorner M \urcorner(\vec{z}, K))\sigma, F(\vec{V})) \\ = & tramp([\ ]);\ cont(\ulcorner M \urcorner(\vec{z}\sigma, K), F(\vec{V})) \\ \longrightarrow & tramp([\ ]);\ (M^\dagger(App(F(\vec{V}), K)))\{(\vec{z}\sigma)/\vec{z}\} \\ = & tramp([\ ]);\ (M^\dagger k)\{App(F(\vec{V}), K)/k,\ (\vec{z}\sigma)/\vec{z}\} \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ cont(App(F(\vec{V}), K), W) \\ \longrightarrow & tramp([\ ]);\ apply(F(\vec{V}), W, K) \end{array}$$

  Applying the inductive hypothesis twice, we get

  $$L\sigma^\bullet \Downarrow_{\mathsf{s}}\ (F(\vec{V}))^\bullet = \lambda^{\mathsf{s}} x.N\{\vec{V}^\bullet/\vec{y}\} \text{ and}$$
  $$M\sigma^\bullet \Downarrow_{\mathsf{s}} W^\bullet$$

  Now we show the third leg of Beta, namely $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_a V^\bullet$, by taking cases on a.

  If $a = \mathsf{s}$ then N is such that $(F(\vec{y}) \Rightarrow (N^\dagger k)\{arg/x\})) \in \text{cases}(apply, \mathcal{S})$. So the reduction continues as

  $$\begin{array}{rl} & tramp([\ ]);\ apply(F(\vec{V}), W, k) \\ \longrightarrow & tramp([\ ]);\ (N^\dagger k)\{\vec{V}/\vec{y}, W/x\} \\ & = (N^\dagger k)\{\vec{V}/\vec{y}\}\{W/x\} \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ cont(k, V) \end{array}$$

  So by the inductive hypothesis (invoking Lemma 2),

  $$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{s}} V^\bullet.$$

  If $a = \mathsf{c}$ then N is such that

  $$(F(\vec{y}) \Rightarrow N^*\{arg/x\}) \in \text{cases}(apply, \mathcal{C}) \text{ and}$$
  $$(F(\vec{y}) \Rightarrow Call(F(\vec{y}), arg, k)) \in \text{cases}(apply, \mathcal{S}).$$

  So the reduction continues as:

  $$\begin{array}{rl} & tramp([\ ]);\ apply(F(\vec{V}), W, k) \\ \longrightarrow & tramp([\ ]);\ Call(F(\vec{V}), W, k) \\ \longrightarrow & tramp(Call(F(\vec{V}), W, k)) \\ \longrightarrow & tramp(\mathsf{req}\ cont\ (k, apply(F(\vec{V}), W))) \\ \longrightarrow & tramp(\mathsf{req}\ cont\ (k, N^*\{\vec{V}/\vec{y}, W/x\})) \\ = & tramp(\mathsf{req}\ cont\ (k, N^*\{\vec{V}/\vec{y}\}\{W/x\})) \\ \longrightarrow\!\!\!\!\rightarrow & tramp(\mathsf{req}\ cont\ (k, V)) \\ \longrightarrow\!\!\!\!\rightarrow & tramp([\ ]);\ cont(k, V) \end{array}$$

  So by the inductive hypothesis (invoking Lemma 2),

  $$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{s}} V^\bullet.$$

  The judgment $(LM)\sigma^\bullet \Downarrow_{\mathsf{s}} V^\bullet$ follows by Beta. *huzzah!*

- Case V for (i) and (ii). Write W for M, which must also be a value.

  Because the starting term is a value, the reduction is of zero steps: In the client case: $M^*\sigma = V \longrightarrow\!\!\!\!\rightarrow V$. We have that $M^* = W^\circ$ so $W^\circ\sigma = V$.

  In the server case: $(M^\dagger K)\sigma = cont(K, V) \longrightarrow\!\!\!\!\rightarrow cont(K, V)$. We have that $M^\dagger K = cont(K, W^\circ)$ so $W^\circ\sigma = V$.

  Using the substitution lemma (Lemma 4) and the inverseness of $(-)^\bullet$ to $(-)^\circ$, we get $(W^\circ\sigma)^\bullet = W\sigma^\bullet$. Now $(W^\circ\sigma)^\bullet = V^\bullet$ so $V^\bullet = M\sigma^\bullet$. And so the reduction $M\sigma^\bullet \Downarrow_a V$ follows by Value. *huzzah!* □

We turn to the completeness of the translation. First we show that continuations K in λ_{cs} are closely related to evaluation contexts E in λ_{rpc}. Using this we show the possible forms of λ_{cs} terms that map to λ_{rpc} application terms.

Lemma 6. *Given definition-sets $\mathcal{C}, \mathcal{S}$ and a continuation K, one of the following holds:*

(a) The form of $K^{\$}_{\mathcal{C},\mathcal{S}}$ is $[\ ]$ and $K = k$.

(b) The form of $K^{\$}_{\mathcal{C},\mathcal{S}}$ is VE and there exist J, V' such that

$$\begin{array}{rcl} K & = & J\{App(V', k)/k\}, \\ V'^\bullet_{\mathcal{C},\mathcal{S}} & = & V \quad \textit{and} \\ J^{\$}_{\mathcal{C},\mathcal{S}} & = & E. \end{array}$$

(c) The form of $K^{\$}_{\mathcal{C},\mathcal{S}}$ is EM and there exist $J, M', F, \vec{V}$ such that

$$\begin{aligned} K &= J\{F(\vec{V},k)/k\}, \\ (F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) &\in \text{cases}(cont,\mathcal{S}), \\ (M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathcal{C},\mathcal{S}} &= M \quad \textit{and} \\ J^{\$}_{\mathcal{C},\mathcal{S}} &= E. \end{aligned}$$

PROOF. The proof is by induction on K. Take cases on its form:

- Case k. By def., $K^{\$}_{\mathcal{C},\mathcal{S}} = [\ ]$, proving (a). *huzzah!*
- Case $App(U, K')$. By definition, $K^{\$}_{\mathcal{C},\mathcal{S}} = K'^{\$}_{\mathcal{C},\mathcal{S}}[U^{\bullet}_{\mathcal{C},\mathcal{S}}[\ ]]$.

  If $K' = k$, then we prove (b). By def., $K'^{\$}_{\mathcal{C},\mathcal{S}} = [\ ]$. Letting $J = k$ and $E = [\ ]$ we get $K = J\{App(U,k)/k\}$ and $J^{\$}_{\mathcal{C},\mathcal{S}} = E$ as needed.

  If K' is not k then the induction hypothesis gives us one of the cases (b) or (c); we prove the same case. The IH provides J' and E' with $J'^{\$}_{\mathcal{C},\mathcal{S}} = E'$. Let $J = App(U, J')$ and $E = E'[U^{\bullet}_{\mathcal{C},\mathcal{S}}[\ ]]$. By def., $J^{\$}_{\mathcal{C},\mathcal{S}} = E$. The required relation between K and J follows by manipulation of substitutions. The other needed items (V', or F and M') carry through from the inductive hypothesis. *huzzah!*
- Case $G(\vec{W}, K')$. By definition of $(-)^{\$}_{\mathcal{C},\mathcal{S}}$, we have N such that

  $$(G(\vec{y},k) \Rightarrow N^{\dagger}(App(arg,k))) \in \text{cases}(cont,\mathcal{S})$$

  which gives us $K^{\$}_{\mathcal{C},\mathcal{S}} = K'^{\$}_{\mathcal{C},\mathcal{S}}[[\ ](N\{\vec{W}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\})]$.

  If $K' = k$, then we prove (c). By def., $K'^{\$}_{\mathcal{C},\mathcal{S}} = [\ ]$. Let F be G. Letting $J = k$ and $E = [\ ]$ we get $K = J\{G(\vec{W},k)/k\}$ and $J^{\$}_{\mathcal{C},\mathcal{S}} = E$ as needed. Let M' be $N^{\dagger}k$. Then $(M'\{\vec{W}/\vec{y}\}))^{\ddagger}_{\mathcal{C},\mathcal{S}} = M'^{\ddagger}_{\mathcal{C},\mathcal{S}}\{\vec{W}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\}) = N\{\vec{W}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\}$ as needed.

  If K' is not k then the induction hypothesis gives us one of the cases (b) or (c); we prove the same case. The IH provides J' and E' with $J'^{\$}_{\mathcal{C},\mathcal{S}} = E'$. Let $J = G(\vec{W}, J')$ and $E = E'[[\ ](N\{\vec{W}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\})]$. By def., $J^{\$}_{\mathcal{C},\mathcal{S}} = E$. The required relation between K and J follows by manipulation of substitutions. The other needed items (V', or F and M') carry through from the inductive hypothesis. *huzzah!* □

LEMMA 7 (APPLICATIONS' $(-)^{\ddagger}$-PREIMAGE). *Given a λ_{cs} term N' and λ_{rpc} terms L and M with $N'^{\ddagger}_{\mathcal{C},\mathcal{S}} = LM$, then at least one of the following hold:*

(a) there exist λ_{cs} terms L', M', $\vec{V}$ and name F s.t.:

$$\begin{aligned} N' &= L'\{F(\vec{V},k)/k\}, \\ L'^{\ddagger}_{\mathcal{C},\mathcal{S}} &= L \\ (M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathcal{C},\mathcal{S}} &= M \quad \textit{and} \\ (F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) &\in \text{cases}(cont,\mathcal{S}). \end{aligned}$$

(b) L is a value and there exist λ_{cs} terms V' and M' s.t.:

$$\begin{aligned} N' &= M'\{App(V',k)/k\}, \\ V'^{\bullet}_{\mathcal{C},\mathcal{S}} &= L \quad \textit{and} \\ M'^{\ddagger}_{\mathcal{C},\mathcal{S}} &= M. \end{aligned}$$

(c) L and M are values and there exist λ_{cs} terms V' and W' s.t.:

$$\begin{aligned} &N' = apply(V', W', k) \textit{ and} \\ &V'^{\bullet}_{\mathcal{C},\mathcal{S}} = L \textit{ and } W'^{\bullet}_{\mathcal{C},\mathcal{S}} = M. \end{aligned}$$

PROOF. Define two terms, K and Q, as follows: Consider the possible forms of N': either $cont(K, U')$ or $apply(U', W', K)$. In the first case, let $Q = U'^{\bullet}_{\mathcal{C},\mathcal{S}}$, and in the other let $Q = U'^{\bullet}_{\mathcal{C},\mathcal{S}}W'^{\bullet}_{\mathcal{C},\mathcal{S}}$. In each case, by def., $N'^{\ddagger}_{\mathcal{C},\mathcal{S}} = K^{\$}_{\mathcal{C},\mathcal{S}}[Q]$.

Take cases on the structure of $K^{\$}_{\mathcal{C},\mathcal{S}}$ as enumerated by Lemma 6.

- Case $K^{\$}_{\mathcal{C},\mathcal{S}} = [\ ]$. We show consequent (c).

  Take cases on the form of N':

  - Case $cont(K, U')$. Here $N'^{\ddagger}_{\mathcal{C},\mathcal{S}} = U'^{\bullet}_{\mathcal{C},\mathcal{S}}$; but this is not an application, a contradiction. *huz.*
  - Case $apply(U', W', K)$

    Here $N'^{\ddagger}_{\mathcal{C},\mathcal{S}} = U'^{\bullet}_{\mathcal{C},\mathcal{S}}W'^{\bullet}_{\mathcal{C},\mathcal{S}} = LM$. By structural equality, then, $U'^{\bullet}_{\mathcal{C},\mathcal{S}} = L$ and $W'^{\bullet}_{\mathcal{C},\mathcal{S}} = M$. *huzzah!*
- Case $K^{\$}_{\mathcal{C},\mathcal{S}} = VE$. We show consequent (b). We have $L = V$ and $E[Q] = M$. From Lemma 6 we have J and V' such that $K = J\{App(V', k)/k\}$ with $J^{\$}_{\mathcal{C},\mathcal{S}} = E$ and $V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$. Let M' be the one of $cont(J, U')$ or $apply(U', W', J)$ that matches the form of N'. Then $N' = M'\{App(V,k)/k\}$. Calculate that $M'^{\ddagger}_{\mathcal{C},\mathcal{S}} = J^{\$}_{\mathcal{C},\mathcal{S}}[Q] = E[Q] = M$, as needed. *huzzah!*
- Case $K^{\$}_{\mathcal{C},\mathcal{S}} = EN$. We show consequent (a). We have $E[Q] = L$ and $N = M$. From Lemma 6 we have terms J, M' and $\vec{V}$ and name F so that $K = J\{F(\vec{V},k)/k\}$,

  $$(F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) \in \text{cases}(cont,\mathcal{S}),$$

  and $(M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathcal{C},\mathcal{S}} = M$ and $J^{\$}_{\mathcal{C},\mathcal{S}} = E$; this supplies the needed F, $\vec{V}$ and M'. Let L' be the one of $cont(J, U')$ or $apply(U', W', J)$ that matches the form of N'. Then $N' = L'\{F(\vec{V},k)/k\}$, as needed. Calculate that $L'^{\ddagger}_{\mathcal{C},\mathcal{S}} = J^{\$}_{\mathcal{C},\mathcal{S}}[Q] = E[Q] = L$, as needed. *huzzah!* □

NOTATION 1. *Write $M \twoheadrightarrow_{\mathcal{C},\mathcal{S}} V$ for*

$$tramp([\ ]); M \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} tramp([\ ]); cont(k, V).$$

The next lemma shows that the behavior of terms in λ_{cs} follows that of the corresponding λ_{rpc} terms.

LEMMA 8 (COMPLETENESS). *Given any λ_{cs} terms M, V and definitions $\mathcal{C}$ and $\mathcal{S}$,*

(i) If $M^{\star}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{c}} V$ then there exists V' with $V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$ and $M \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} V'$, and

(ii) if $M^{\ddagger}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{s}} V$ then there exists V' with $V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$ and $M \twoheadrightarrow_{\mathcal{C},\mathcal{S}} V'$

PROOF. By induction on the derivation of the given $M^{\star}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{c}} V$ or $M^{\ddagger}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{s}} V$. Take cases on the final step of the derivation:

- Case VALUE. The low-level reduction is of zero steps. The initial low-level term must be a value, V', since its image under the reverse translation is a value. The initial and final low-level terms are the same because ${V'}^{\star}_{\mathcal{C},\mathcal{S}} = {V'}^{\bullet}_{\mathcal{C},\mathcal{S}}$ and ${V'}^{\ddagger}_{\mathcal{C},\mathcal{S}} = {V'}^{\bullet}_{\mathcal{C},\mathcal{S}}$ on values. *huzzah!*
- Case BETA. Recall the rule:

$$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{LM \Downarrow_a V}$$

Take cases on a, the location where the reduction takes place.

– Case $a = \mathsf{c}$.
Because the starting λ_{cs} term maps to LM under $(-)^{\star}$, it must be of the form $\mathit{apply}(L', M')$ with ${L'}^{\star}_{\mathcal{C},\mathcal{S}} = L$ and ${M'}^{\star}_{\mathcal{C},\mathcal{S}} = M$.
By IH we have normal forms

$$L' \longrightarrow\!\!\!\!\!\rightarrow F(\vec{V}) \quad \text{and} \quad M' \longrightarrow\!\!\!\!\!\rightarrow W'$$

satisfying

$$(F(\vec{V}))^{\bullet}_{\mathcal{C},\mathcal{S}} = \lambda^b x.N \quad \text{and} \quad {W'}^{\bullet}_{\mathcal{C},\mathcal{S}} = W$$

So the term reduces as follows:

$$\mathit{apply}(L', M') \quad \longrightarrow\!\!\!\!\!\rightarrow \quad \mathit{apply}(F(\vec{V}), W')$$

To finish the reduction, take cases on b.
If b is c then we have N' such that

$$(F(\vec{y}) \Rightarrow N') \in \text{cases}(\mathit{apply}, \mathcal{C})$$

Therefore

$$\begin{aligned}
& (N'\{x/arg\})^{\star}_{\mathcal{C},\mathcal{S}}\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} = N \text{(def. of } (F(\vec{V}))^{\bullet}) \\
& (N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\star}_{\mathcal{C},\mathcal{S}} = N \\
& (N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\star}_{\mathcal{C},\mathcal{S}}\{{W'}^{\bullet}_{\mathcal{C},\mathcal{S}}/x\} \\
= \;& N\{W/x\} \\
= \;& (N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\})^{\star}_{\mathcal{C},\mathcal{S}}
\end{aligned}$$

And so by IH

$$N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\} \longrightarrow\!\!\!\!\!\rightarrow V'$$
with $V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$.

Now we can finish the reduction:

$$\begin{aligned}
& \mathit{apply}(F(\vec{V}), W') \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}(\mathsf{req}\ \mathit{apply}\ (F(\vec{V}), \mathit{arg}, \mathit{Fin}())) \\
\longrightarrow \;& \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}), \mathit{arg}, \mathit{Fin}()) \\
\longrightarrow \;& \mathit{tramp}([\ ]);\ N'\{\vec{V}/\vec{y}\}\{W'/\mathit{arg},\ \mathit{Fin}()/k\} \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}([\ ]);\ \mathit{cont}(\mathit{Fin}(), V') \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}(\mathit{Return}(V')) \\
\longrightarrow \;& V'
\end{aligned}$$

which was to be shown. *huzzah!*

– Case $a = \mathsf{s}$. Let X be the term such that $X^{\ddagger}_{\mathcal{C},\mathcal{S}} = LM$. Lemma 7 nominates the possible forms of X.
First consider the case of Lemma 7(a). This gives us terms L' and M' such that

$$\begin{aligned}
X &= L'\{G(\vec{U}, k)/k\} \\
{L'}^{\ddagger}_{\mathcal{C},\mathcal{S}} &= L, \\
(M'\{\vec{U}/\vec{z}\})^{\ddagger}_{\mathcal{C},\mathcal{S}} &= M \text{ and} \\
(G(\vec{z}, k) \Rightarrow M'\{\mathit{App}(\mathit{arg}, k)/k\}) &\in \text{cases}(\mathit{cont}, \mathcal{S}).
\end{aligned}$$

By IH we have these normal forms:

$$L' \twoheadrightarrow_{\mathcal{C},\mathcal{S}} F(\vec{V}) \quad \text{and} \quad M'\{\vec{U}/\vec{y}\} \twoheadrightarrow_{\mathcal{C},\mathcal{S}} W'$$

satisfying

$$(F(\vec{V}))^{\bullet}_{\mathcal{C},\mathcal{S}} = \lambda^b x.N \quad \text{and} \quad {W'}^{\bullet}_{\mathcal{C},\mathcal{S}} = W$$

And so we can trace the reduction of our term:

$$\begin{aligned}
& \mathit{tramp}([\ ]);\ L'\{G(\vec{U}, k)/k\} && \text{(a)} \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}([\ ]);\ \mathit{cont}(G(\vec{U}, k), F(\vec{V})) \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}([\ ]);\ M'\{\mathit{App}(F(\vec{V}), k)/k\} && \text{(b)} \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}([\ ]);\ \mathit{cont}(\mathit{App}(F(\vec{V}), k), W') \\
\longrightarrow\!\!\!\!\!\rightarrow \;& \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}), W', k) && \text{(c)}
\end{aligned}$$

To finish the reduction, take cases on b.
If b is c then

$$\begin{aligned}
& (F(\vec{y}) \Rightarrow \mathit{Call}(F(\vec{y}), \mathit{arg}, k)) \in \text{cases}(\mathit{apply}, \mathcal{S}) \\
\text{and} \quad & (F(\vec{y}) \Rightarrow N') \in \text{cases}(\mathit{apply}, \mathcal{C})
\end{aligned}$$

Therefore

$$\begin{aligned}
& (N'\{x/arg\})^{\star}_{\mathcal{C},\mathcal{S}}\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} = N \text{(def. of } (F(\vec{V}))^{\bullet}) \\
& (N'\{\vec{V}/\vec{y}\}\{x/arg\})^{\star}_{\mathcal{C},\mathcal{S}} = N \\
& (N'\{\vec{V}/\vec{y}\}\{x/arg\})^{\star}_{\mathcal{C},\mathcal{S}}\{{W'}^{\bullet}_{\mathcal{C},\mathcal{S}}/x\} \\
= \;& N\{W/x\} \\
= \;& (N'\{\vec{V}/\vec{y}\}\{x/arg\}\{W'_{\mathcal{C},\mathcal{S}}/x\})^{\star}_{\mathcal{C},\mathcal{S}}
\end{aligned}$$

And so by IH

$$N'\{\vec{V}/\vec{y}\}\{x/arg\}\{W'_{\mathcal{C},\mathcal{S}}/x\} \longrightarrow\!\!\!\!\!\rightarrow V'$$
with $V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$

Now we can finish the reduction:

$$\begin{array}{rl} & \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}), W', k) \\ \longrightarrow\!\!\!\!\rightarrow & \mathit{tramp}([\ ]);\ \mathit{Call}(F(\vec{V}), W', k) \\ \longrightarrow & \mathit{tramp}(\mathit{Call}(F(\vec{V}), W', k)) \\ \longrightarrow\!\!\!\!\rightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k, \mathit{apply}(F(\vec{V}), W'))) \\ \longrightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k, N\{\vec{V}/\vec{y}, W'/\mathit{arg}\})) \\ \longrightarrow\!\!\!\!\rightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k, V')) \\ \longrightarrow & \mathit{tramp}([\ ]);\ \mathit{cont}(k, V') \end{array}$$

which was to be shown.

If b is s then we have N' such that

$$(F(\vec{y}) \Rightarrow N') \in \mathrm{cases}(\mathit{apply}, \mathcal{S})$$

Therefore

$$\begin{array}{rcl} (N'\{x/\mathit{arg}\})^{\ddagger}_{\mathcal{C},\mathcal{S}}\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} & = & N(\text{def. of } (F(\vec{V}))^{\bullet}) \\ (N'\{x/\mathit{arg}\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathcal{C},\mathcal{S}} & = & N \\ & & (N'\{x/\mathit{arg}\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathcal{C},\mathcal{S}}\{W'^{\bullet}_{\mathcal{C},\mathcal{S}}/x\} \\ & = & N\{W/x\} \\ & = & (N'\{x/\mathit{arg}\}\{\vec{V}/\vec{y}\}\{W'/x\})^{\ddagger}_{\mathcal{C},\mathcal{S}} \end{array}$$

And so by IH

$$N'\{x/\mathit{arg}\}\{\vec{V}/\vec{y}\}\{W'/x\} \quad \twoheadrightarrow_{\mathcal{C},\mathcal{S}} V' \quad \text{with } V'^{\bullet}_{\mathcal{C},\mathcal{S}} = V$$

Now we can finish the reduction:

$$\begin{array}{rl} & \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}), W', k) \\ \longrightarrow\!\!\!\!\rightarrow & \mathit{tramp}([\ ]);\ N'\{\vec{V}/\vec{y}, W'/\mathit{arg}\} \\ \longrightarrow & \mathit{tramp}([\ ]);\ \mathit{cont}(k, V') \end{array}$$

which was to be shown.

Now consider the other cases from Lemma 7, either (b) or (c). Then we use the above reduction sequence but beginning from the correspondingly marked line. *huzzah!* □

At last we can state the full correctness result concisely:

PROPOSITION 1. *For any closed λ_{rpc} term M, value V and definitions $(\mathcal{C}, \mathcal{S}) = ([\![M]\!]^{\mathsf{c},\mathrm{top}}, [\![M]\!]^{\mathsf{s},\mathrm{top}})$,*

$$M \Downarrow_{\mathsf{c}} V \iff \textit{exists } V' \textit{ s.t. } M^* \twoheadrightarrow_{\mathcal{C},\mathcal{S}} V' \textit{ and } V'^{\bullet} = V$$

PROOF. The ($\Leftarrow$) implication is immediate from Lemma 5. To infer the ($\Rightarrow$) implication from Lemma 8 we need to show that the given M has an M' such that $M'^{\star}_{\mathcal{C},\mathcal{S}} = M$. We can construct $M' = M^*$ and the needed relationship follows directly from the retraction lemma. □

4. A RICHER CALCULUS

The calculus $\lambda_{\langle\rangle}$ in Figure 9 adds location brackets $\langle\cdot\rangle^a$ to λ_{rpc} and allows *unannotated* λ-abstractions. The interpretation of a bracketed expression $\langle M\rangle^a$ in a location-b context is a computation that evaluates every computation step lexically within M at location a and returns the value to the location b. Unannotated λ-abstractions are not treated as values: we want all values to be mobile, and yet the body of an unannotated abstraction should inherit its required location from the surrounding lexical context. Thus, to become a value, the abstraction itself must become tagged with this location, and the ABSTR rule attaches this annotation when it is not already provided.

Syntax

$$\begin{array}{rrcl} \text{constants} & c & & \\ \text{variables} & x & & \\ \text{locations} & a,\ b & & \\ \text{terms} & L, M, N & ::= & \langle M\rangle^a \mid \lambda x.N \mid LM \mid V \\ \text{values} & V, W & ::= & \lambda^a x.N \mid x \mid c \end{array}$$

Semantics

$$\boxed{M \Downarrow_a V}$$

$$V \Downarrow_a V \qquad \text{(VALUE)}$$

$$\lambda x.N \Downarrow_a \lambda^a x.N \qquad \text{(ABSTR)}$$

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

$$\frac{M \Downarrow_b V}{\langle M\rangle^b \Downarrow_a V} \qquad \text{(CLOTHE)}$$

Figure 9: The bracket-located lambda calculus, $\lambda_{\langle\rangle}$.

$$\begin{array}{rcll} [\![\langle M\rangle^b]\!]_a & = & (\lambda^b x.[\![M]\!]^b)() & x \text{ fresh} \\ [\![\lambda x.N]\!]_a & = & \lambda^a x.[\![N]\!]^a & \\ [\![\lambda^b x.N]\!]_a & = & \lambda^b x.[\![N]\!]^b & \\ [\![x]\!]_a & = & x & \\ [\![c]\!]_a & = & c & \end{array}$$

Figure 10: Translation from $\lambda_{\langle\rangle}$ to $\lambda_{\mathbf{rpc}}$.

Figure 10 gives a translation from $\lambda_{\langle\rangle}$ to λ_{rpc}. Bracketed terms $\langle M\rangle^a$ are simply treated as applications of located thunks; and as expected, unannotated abstractions $\lambda x.N$ inherit their annotation from their lexical context.

The relationship between this calculus and the λ_{cs} calculus is looser than the one previously shown. The forward translation this time is not injective, so there can be no exact reverse translation as before. (For example, $\langle M\rangle^b$ and $(\lambda^b x.M)()$ go to the same term.) As a result, we would need to use a simulation relation.

Location brackets such as these may be an interesting language feature, allowing programmers to designate the location of computation of arbitrary terms.

5. RELATED WORK

Location-aware languages.

Although this the first work we're aware of that shows how to implement a location-aware language on top of a stateless server model, there is much work in the theory and implementation of location-aware languages.

Lambda 5 [15, 14] is a location-aware calculus with con-

structs for controlling the location and movement of terms and values. Lambda 5 offers fine control over these runtime movements, whereas our calculus uses the usual scope discipline of λ-binding and is profligate with data movements. Like ours, the translation of Lambda 5 to an operational model also involves a CPS translation; and where we have used defunctionalization, it uses the similar technique of closure conversion; it uses a stateful-server approach and hence no trampoline is necessary.

Neubauer and Thiemann [17] give an algorithm for splitting a location-annotated sequential program into separate concurrent programs that communicate over channels. By default the system requires that the two systems be equal peers, rather than an asymmetrical client-server pair. They note that "Our framework is applicable to [the special case of a web application] given a suitable mediator that implements channels on top of HTTP." The trampoline technique we have given provides such a mediator. They use session types to show that the various processes' use of channels are type-correct over the course of the interaction, in contrast to the always-receptive, stateless server of the present work.

Ohori and Kato [19] describe a locative language where the program is separated by the programmer into files for locations and locations can *import* values via a global name table. They translate this to a lower-level language with explicit RPC calls, which are restricted to act on concrete types communicable over a network. Like Neubauer and Thiemann's work, and unlike ours, at the low level the locations are all mutually accessible.

In a security context, Zdancewic, et al. [24] developed a calculus with brackets, which served as the model for our $\lambda_{\langle\rangle}$. Their results show how a type discipline, with type translations taking place at the brackets, can be used to prove that certain principals (analogous to locations) cannot inspect certain values passed across an interface. Such a discipline could be applied to our calculus, to address information-flow security between client and server. Matthews and Findler [12] give a nearly identical semantics which models multi-language programs; here languages act like principals or locations in the other systems.

Defunctionalization.

After first being introduced in a lucid but informal account by John Reynolds [22], defunctionalization has been formalized and verified in a typed setting in several papers [3, 2, 21, 18, 1]. We have formalized defunctionalization in an untyped setting, which is slightly easier because we need not segregate the application machinery by type. Danvy and Nielsen [8] and Danvy and Millikin [7] explore a number of uses and properties of defunctionalization.

Defunctionalization is very similar to lambda-lifting [11], but lambda-lifting does not reify a closure as an inspectable value. Thus it would not be applicable here, where we need to serialize the function to send across the wire.

Murphy [14] uses closure-conversion in place of our defunctionalization; the distinction here is that the converted closures still contain code pointers, rather than using a stable name to identify each abstraction. These code pointers are only valid as long as the server is actively running, and thus it may be difficult to achieve statelessness with such an approach.

Continuation-Passing.

The continuation-passing transformation has a long and storied history [23], going back to the 1970s [9, 20].

Trampolined style.

Ganz et al. [10] introduced the *trampolined style* of tail-form programs, whereby every tail call is replaced with the construction of a value containing a thunk for the tail call. Instead of performing the call, then, the program is returning a representation of the next tail call to be made. The program is then to be invoked from a loop, called the trampoline, which might treat the thunk in various ways, perhaps invoking it immediately, interjecting other actions, juggling several thunks or other possibilities. A program in trampolined style only does a bounded amount of work before returning the next thunk. The authors give a translation taking any program in *tail form* (which includes CPS) to one in trampolined style.

The system presented here is an instance of trampolined style in the sense that each remote call from the client is wrapped in a trampoline, and all remote calls from the server to the client are transformed to trampoline bounces. The fact that local function calls take place directly is a departure from earlier work.

Extensions.

Corcoran, et al. [6] have extended the location-aware language Links with a type system that identifies data items with security policies, and ensures statically that the policies are not violated by the program's runtime behavior, in view of the low-level locative semantics of Links.

6. CONCLUSIONS AND FUTURE WORK

We've shown how to compile a location-aware language to an asymmetrical, stateless, client-server calculus by using a CPS transformation and trampoline to represent the server's call stack as a value on the client. We hope to extend this in several directions.

This work uses a source calculus with location annotations, but writing the annotations may burden the programmer. It may be possible to automatically assign location annotations so as to optimize communication costs, perhaps by applying the work by Neubauer [16]. Because the dynamic location behavior of a program may be hard to predict, and because there are a variety of possible communication and computation cost models, and perhaps other issues to consider, such as security, the problem is multifaceted.

We hope to extend the source calculus by adding language features including exceptions, and generalizing by allowing each annotation to consist of a *set* of permissible locations (rather than a single one). We also hope to implement the $\lambda_{\langle\rangle}$ calculus in Links.

As noted in the introduction, this calculus treats only a simple form of state, namely control state. We might wish to add a store with mutable references, in the fashion of ML. References could be treated as located, for example at the location where they are created. Statelessness could be preserved by serializing the store but encrypting server-located data so that only the server can read them. A security-conscious type system such as that of Corcoran, et al. [6] might be particularly useful here.

Other kinds of state can also be problematic, for example ongoing transactions with other services (disk, databases,

and so on). These are more difficult since they don't admit serialization. Future work might accommodate limited statefulness on the server, with a facility for managing this state.

7. ACKNOWLEDGEMENTS

Sam Lindley and Ian Stark provided invaluable insights and discussions in the development of this work. We also thank the ICFP '08 and '09 and PPDP '09 reviewers.

8. REFERENCES

[1] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *TACS '01*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer, 2001.

[2] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *SIGPLAN Not.*, 32(8):25–37, 1997.

[3] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical report, Oregon Graduate Institute, 1994.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.

[6] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, labeld-based secuirty enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2009. To appear.

[7] Olivier Danvy and Kevin Millikin. Refunctionalization at work. Technical Report RS-08-4, BRICS, June 2008.

[8] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *PPDP '01*, pages 162–174. ACM, 2001.

[9] Michael J. Fischer. Lambda calculus schemata. *SIGACT News*, (14):104–109, 1972.

[10] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *ICFP '99*. ACM Press, September 1999.

[11] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[12] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *POPL '07*, pages 3–10, New York, NY, USA, 2007. ACM.

[13] Jacob Matthews, Robert Bruce Findler, Paul Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. *Automated Software Engineering*, 11:337–364, 10 2004.

[14] Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2007.

[15] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS '04*, pages 286–295, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Matthias Neubauer. *Multi-Tier Programming*. PhD thesis, Universität Freiburg, 2007.

[17] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[18] Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, December 2000.

[19] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *POPL '93*, pages 99–112, New York, NY, USA, 1993. ACM.

[20] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *POPL '04*, pages 89–98, New York, NY, USA, 2004. ACM.

[22] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

[23] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, 1993.

[24] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: a syntactic proof technique. In *ICFP '99*, pages 197–207, New York, NY, USA, 1999. ACM Press.