A located lambda calculus

A located lambda calculus

Ezra elias kilty Cooper Philip Wadler
University of Edinburgh

Abstract

Several recent language designs have offered a unified language for programming a distributed system; we call these "location-aware" languages. These languages provide constructs that allow 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 web engineering insists 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. Thus far, most implementations of unified location-aware languages have ignored this precept, usually keeping a process for each client running on the server, or otherwise storing state information in memory. We show how to implement a location-aware language on top of the stateless-server model.

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 multuple 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 allow the programmer the illusion of a persistent environment encompassing both client and server. 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.

Murphy et al. (2004) introduced a core located calculus, Lambda 5, and Murphy (2007) built upon this a full-fledged programming language, ML5, also showing how to compile it to separate code for client and server. Neubauer and Thiemann (2005) also introduced a variant of ML with location annotations and showed how to perform a splitting transformation to produce code for each location. However, each of these works relied on concurrently-running servers. Ours is the first work we're aware of that shows how to implement a located language on top of a stateless server model.

Our technique involves three essential transformations: defunctionalization *à la* Reynolds, CPS transformation, and a "trampoline" that allows tunnelling server-to-client requests within server responses.

We have implemented a version of this feature in the Links language (Cooper et al. 2006). In the current version, only calls to top-level functions can pass control between client and server. Here we provide a formalization and show how to relax the top level-function restriction. In particular, the current, limited version requires just a CPS translation and a trampoline; defunctionalization is needed in implementing nested annotations.

This paper This paper presents a simple λ-calculus enriched with location annotations, allowing the programmer to indicate the location where a fragment of code should execute. The semantics of this calculus clarifies where each computation step is allowed to take place.

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

As a stepping stone to the full translation, we formally define defunctionalization. By isolating this phase of the larger translation, we hope to clarify the notation and formal techniques that will be used in the full translation. While there are many compact definitions of CPS transformations in the literature, we did not find such a compact and formal definition of defunctionalization. We give a complete formal definition of defunctionalization in eleven lines.

2. Defunctionalization

Source calculus, λ_{src}

Figure 1 shows an entirely pedestrian call-by-value λ-calculus, called λ_{src}, defined with a big-step reduction relation $M \Downarrow V$, stating that term M reduces to value V. We write $N\{V/x\}$ for the capture-avoiding substitution of a value V for the variable x in the term N, and $N\{V_1/x_1, \ldots, V_n/x_n\}$ for the simultaneous capture-avoiding substitution of each V_i for the corresponding x_i. We let σ range over such n-ary substitutions. We assume terms are equal up to α-equivalence.

First-order machine

The defunctionalized machine, DM, defined in Figure 2, is a first-order machine, in contrast to λ_{src} which allowed expressions in the function position of an application. In DM, terms, ranged over by L, M, and N, are first-order; they are built from constants c, variables x, constructor applications $F(\vec{M})$, function applications

Syntax

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

Semantics (big-step reduction)

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

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

Figure 1. Higher-order source, λ_{src}.

Syntax

$$\begin{array}{rrcl} \text{variables} & x, y, z & & \\ \text{function names} & f, g & & \\ \text{constructors} & F, G & & \\ \text{values} & V, W, K & ::= & c \mid x \mid F(\vec{V}) \\ \text{terms} & M, N & ::= & c \mid x \mid f(\vec{M}) \mid \\ & & & F(\vec{M}) \mid \mathsf{case}\ M\ \mathsf{of}\ \mathcal{A} \\ \text{alternative sets} & \mathcal{A} & & \text{a set of } A \text{ items} \\ \text{case alternatives} & A & ::= & F(\vec{x}) \to M \\ \text{eval. contexts} & E & ::= & [\ ] \mid f(\vec{V}, E, \vec{M}) \mid \\ & & & F(\vec{V}, E, \vec{M}) \mid \mathsf{case}\ E\ \mathsf{of}\ \mathcal{A} \\ \text{function def.} & D & ::= & f(\vec{x}) = M \\ \text{definition set} & \mathcal{D} & ::= & \mathsf{letrec}\ D\ \mathsf{and} \cdots \mathsf{and}\ D \end{array}$$

Semantics (small-step reduction)

$$\boxed{M \longrightarrow_{\mathcal{D}} N}$$

$$\begin{array}{rcl} E[f(\vec{V})] & \longrightarrow_{\mathcal{D}} & E[M\{\vec{V}/\vec{x}\}] \\ & & \text{when } (f(\vec{x}) = M) \in \mathcal{D} \\ E[\mathsf{case}\ (F(\vec{V}))\ \mathsf{of}\ \mathcal{A}] & \longrightarrow_{\mathcal{D}} & E[M\{\vec{V}/\vec{x}\}] \\ & & \text{when } (F(\vec{x}) \to M) \in \mathcal{A} \end{array}$$

Figure 2. First-order target, DM (the Defunctionalized Machine).

$f(\vec{M})$ and case expressions $\mathsf{case}\ M\ \mathsf{of}\ \mathcal{A}$. A list $\mathcal{A}$ of case alternatives is well-formed if it uniquely maps each name.

The machine also uses a set $\mathcal{D}$ of function definitions. Each has the form $f(\vec{x}) = M$, defining a function called f taking parameters $\vec{x}$ which are then bound in the function body M. A definition set is well-formed if it uniquely defines each name. The definitions are mutually recursive, so scope of each definition extends throughout all the other definitions as well as the term under evaluation.

The semantics is defined through a small-step reduction relation $M \longrightarrow_{\mathcal{D}} N$ stating that the term M reduces to the term N in the context of definition-set $\mathcal{D}$. The relation $\longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{D}}$ is the reflexive, transitive closure of $\longrightarrow_{\mathcal{D}}$, with the definitions $\mathcal{D}$ held fixed through the reduction sequence.

Defunctionalization

Defunctionalization is a translation from the terms of λ_{src} to the terms and definition-sets of DM; it is defined in Figure 3. From a term M we compute a defunctionalized term $[\![M]\!]$ and a corresponding definition set, $[\![M]\!]^{\text{top}}$. Let *arg* be a special reserved variable name not appearing in the source program. The coll function

$$\begin{array}{rcll} [\![\lambda x.N]\!] & = & \ulcorner \lambda x.N \urcorner(\vec{y}) & \vec{y} = \text{FV}(\lambda x.N) \\ [\![x]\!] & = & x & \\ [\![c]\!] & = & c & \\ [\![LM]\!] & = & apply([\![L]\!], [\![M]\!]) & \end{array}$$

$$\begin{array}{rcll} \text{coll } f\ LM & = & f(LM) \cup \text{coll } f\ L \cup \text{coll } f\ M & \\ \text{coll } f\ \lambda x.N & = & f(\lambda x.N) \cup \text{coll } f\ N & \\ \text{coll } f\ V & = & f(V) & \text{when } V \neq \lambda x.N \end{array}$$

$$\begin{array}{rcll} [\![\lambda x.N]\!]^{\text{fun,aux}} & = & \{\ulcorner \lambda x.N \urcorner(\vec{y}) \to [\![N]\!]\{arg/x\}\} & \\ & & & \text{where } \vec{y} = \text{FV}(\lambda x.N) \\ [\![M]\!]^{\text{fun,aux}} & = & \{\} & \text{when } M \neq \lambda x.N \\ [\![M]\!]^{\text{fun}} & = & \text{coll } [\![-]\!]^{\text{fun,aux}}\ M & \end{array}$$

$$[\![M]\!]^{\text{top}} = \mathsf{letrec}\ apply(fun,\ arg) = \mathsf{case}\ fun\ \mathsf{of}\ [\![M]\!]^{\text{fun}}$$

Figure 3. Defunctionalization.

$$\begin{array}{rcl} [\![F(\vec{V})]\!]_{\mathcal{D}}^{-1} & = & \lambda x.([\![N\{x/arg\}]\!]^{-1})\{[\![\vec{V}]\!]^{-1}/\vec{y}\} \\ & & \text{when } (F(\vec{y}) \to N) \in \text{cases}(apply, \mathcal{D}), \text{ where } x \text{ fresh} \\ [\![x]\!]_{\mathcal{D}}^{-1} & = & x \\ [\![c]\!]_{\mathcal{D}}^{-1} & = & c \\ [\![apply(L, M)]\!]_{\mathcal{D}}^{-1} & = & [\![L]\!]_{\mathcal{D}}^{-1}\ [\![M]\!]_{\mathcal{D}}^{-1} \end{array}$$

Figure 4. An reverse translation for defunctionalization.

is used by $[\![M]\!]^{\text{top}}$ to traverse a term: it applies a function f to each subterm of its argument, collecting the results.

A special feature of our translation is the use of 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. One example of such a function is the one 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 also use a reverse translation, or retraction of $[\![-]\!]$, defined in Figure 4. Here $[\![M]\!]_{\mathcal{D}}^{-1}$ denotes the retraction of a DM term M in the context of definitions $\mathcal{D}$. We lift the reverse translation to substitutions:

$$\text{If } \sigma \text{ is } \{V_1/x_1, \ldots, V_n/x_n\}$$
$$\text{then } [\![\sigma]\!]_{\mathcal{D}}^{-1} \text{ is } \{[\![V_1]\!]_{\mathcal{D}}^{-1}/x_1, \ldots, [\![V_n]\!]_{\mathcal{D}}^{-1}/x_n\}.$$

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

Definition 1. Define a meta-function cases that returns the branches of a given function when that function is defined by case-analysis. Formally, define

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

This completes the definition of the reverse translation $[\![-]\!]^{-1}$.

The central claim of this section is that the defunctionalized machine simulates the source calculus under this translation. We have a few preliminaries to deal with first.

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.

Definition 2 (Definition containment)**.** A definition set $\mathcal{D}$ *contains* $\mathcal{D}'$, written $\mathcal{D} \geqslant \mathcal{D}'$, if

$$\text{cases}(\mathit{apply}, \mathcal{D}) \supseteq \text{cases}(\mathit{apply}, \mathcal{D}').$$

Correctness of the translation from λ_{src} to DM

Finally, we can state and prove the simulation of λ_{src} by DM. First a few preliminaries.

The function $[\![-]\!]_-^{-1}$ inverts the pair of functions $[\![-]\!]$ and $[\![-]\!]^{\text{top}}$, as follows.

Observation 1. If $\mathcal{D} \geqslant [\![M]\!]^{\text{top}}$ then $[\![[\![M]\!]]_{\mathcal{D}}^{-1} = M$. Note that the forward translation replaces each variable bound by a λ-abstraction with the variable arg, and the reverse translation generates fresh names when generating abstractions. This is acceptable since we identify α-equivalent terms.

Lemma 1 (Substitution-Reverse translation)**.** *The reverse translation preserves substitutions. Given definition-set $\mathcal{D}$, term M and value V, if $\mathcal{D} \geqslant [\![M]\!]^{\text{top}}$, then we have $[\![M\{V/x\}]\!]_{\mathcal{D}}^{-1} = [\![M]\!]_{\mathcal{D}}^{-1}\{[\![V]\!]_{\mathcal{D}}^{-1}/x\}$*

Proof. By induction on the structure of M. The proof of each case is a simple matter of pushing the substitutions down through the terms, applying the inductive hypothesis, and pulling them back up the terms. □

Proposition 1 (Simulation)**.** *Given a λ_{src} term M, DM value V and definitions $\mathcal{D}$ with $\mathcal{D} \geqslant [\![M]\!]^{\text{top}}$, If $[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow V$ then $M \Downarrow [\![V]\!]_{\mathcal{D}}^{-1}$.*

Proof. This follows directly from the following lemma. □

Lemma 2. *Given a λ_{src} term M and a DM value V, substitution σ and definitions $\mathcal{D}$ with $\mathcal{D} \geqslant [\![M]\!]^{\text{top}}$,*

$$\textit{if } [\![M]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow V \textit{ then } M[\![\sigma]\!]_{\mathcal{D}}^{-1} \Downarrow [\![V]\!]_{\mathcal{D}}^{-1}$$

Proof. By induction on the length of the reduction $[\![M]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow V$.

We take cases on the structure of the term M.

- Case V. The reduction is of zero steps, $[\![V]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow [\![V]\!]\sigma$, and $[\![[\![V]\!]\sigma]\!]_{\mathcal{D}}^{-1} = V[\![\sigma]\!]_{\mathcal{D}}^{-1}$. The judgment $V[\![\sigma]\!]_{\mathcal{D}}^{-1} \Downarrow V[\![\sigma]\!]_{\mathcal{D}}^{-1}$ is by VALUE. *huzzah!*
- Case LM. By hypothesis, $[\![LM]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow V$. Recall that

  $$[\![LM]\!]\sigma = \mathit{apply}([\![L]\!]\sigma,\ [\![M]\!]\sigma).$$

  It must be the case that $[\![L]\!]\sigma$ and $[\![M]\!]\sigma$ each reduce to some value, for if not the reduction would get stuck. Further, the value to which $[\![L]\!]\sigma$ reduces must be a constructor application $F(\vec{V})$, for the same reason. Let W be the value to which $[\![M]\!]\sigma$ reduces. Let x be a fresh variable and let N be such that $(F(\vec{y}) \to [\![N]\!]\{\mathit{arg}/x\}) \in \text{cases}(\mathit{apply}, \mathcal{D})$. We know the body of the case for F has the form $[\![N]\!]\{\mathit{arg}/x\}$ because it is in the image of the translation $[\![-]\!]^{\text{top}}$. As such we have

  $$[\![F(\vec{V})]\!]_{\mathcal{D}}^{-1} = \lambda x.N\{[\![\vec{V}]\!]_{\mathcal{D}}^{-1}/\vec{y}\}.$$

  The reduction follows:

  $$\begin{array}{lll} & \mathit{apply}([\![L]\!]\sigma,\ [\![M]\!]\sigma) & \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathit{apply}(F(\vec{V}),\ W) & \\ \longrightarrow & [\![N]\!]\{\vec{V}/\vec{y}, W/x\} & \text{where } \vec{y} = \text{FV}(N) \\ \longrightarrow\!\!\!\!\!\rightarrow & V & \end{array}$$

  Now we apply the inductive hypothesis three times. Note that all of these reductions are no longer than the present one, and the definitions in $[\![L]\!]^{\text{top}}$, $[\![M]\!]^{\text{top}}$ and $[\![N]\!]^{\text{top}}$ are all contained in $\mathcal{D}$ (we know $[\![N]\!]^{\text{top}}$ is contained because $[\![N]\!]$ appeared as the body of a case in apply, and so $\lambda x.N$ must have been a subterm of LM), so the inductive hypothesis applies.

  From $[\![L]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow F(\vec{V})$ we get

  $$L[\![\sigma]\!]_{\mathcal{D}}^{-1} \Downarrow \lambda x.N\{[\![\vec{V}]\!]_{\mathcal{D}}^{-1}/\vec{y}\}.$$

  From $[\![M]\!]\sigma \longrightarrow\!\!\!\!\!\rightarrow W$ we get

  $$M[\![\sigma]\!]_{\mathcal{D}}^{-1} \Downarrow [\![W]\!]_{\mathcal{D}}^{-1}.$$

  From $[\![N]\!]\{\vec{V}/\vec{y}, W/x\} \longrightarrow\!\!\!\!\!\rightarrow V$ we get

  $$N\{[\![\vec{V}]\!]_{\mathcal{D}}^{-1}/\vec{y}, [\![W]\!]^{-1}/x\} \Downarrow [\![V]\!]^{-1}.$$

  By the freshness of x, $N\{[\![\vec{V}]\!]_{\mathcal{D}}^{-1}/\vec{y}\}\{[\![W]\!]_{\mathcal{D}}^{-1}/x\} \Downarrow [\![V]\!]_{\mathcal{D}}^{-1}$. The judgment $(LM)[\![\sigma]\!]_{\mathcal{D}}^{-1} \Downarrow [\![V]\!]_{\mathcal{D}}^{-1}$ follows by BETA. *huzzah!*

□

Syntax

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

Semantics (big-step reduction)

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

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

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

Figure 5. The located lambda calculus, λ_{loc}.

3. The Located λ-calculus

The Located λ-calculus

The located lambda calculus, λ_{loc}, is defined in Figure 5. This calculus extends the pedestrian calculus of Section 2 by tagging λ-abstractions with a location; we use the set of locations $\{\mathsf{c}, \mathsf{s}\}$, because we are interested in the client-server setting, but the calculus would be undisturbed by any other choice of location set.

The annotation on a λ-abstraction indicates the location where its body must execute. Thus an abstraction $\lambda^{\mathsf{c}} x.N$ represents a function that, when applied, would evaluate the term N at the client (c), binding the variable x to the function argument as usual. Constants are assumed to be universal, that is, all locations treat them the same way, and they contain no location annotations.

The semantics, defined in big-step style, uses a judgment of the form $M \Downarrow_a V$, read "the term M, evaluated at location a, results in value V." The reader can verify that the 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.

Again we write $N\{V/x\}$ for the capture-avoiding substitution of a value V for the variable x in the term N. We assume terms are equal up to α-equivalence. The annotation a on $\lambda^a x.N$ has no effect on the binding behavior of names.

3.1 Client/server machine

Our target abstract machine models a pair of interacting agents, a client and a server, that are each first-order computing machines. Figure 6 defines the machine, called CSM.

Being a first-order machine, the application form $f(\vec{M})$ is n-ary and allows only a function name, f, in the function position. The machine 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 machine 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.

Syntax

function names	f, g		
constructor names	F, G		
values	V, W, K	$::=$	$x \mid c \mid F(\vec{V})$
terms	L, M, N	$::=$	$x \mid c \mid f(\vec{M})$
			$\mid F(\vec{M}) \mid \mathsf{case}\ M\ \mathsf{of}\ \mathcal{A}$
			$\mid \mathsf{req}\ f\ (\vec{M})$
alternative sets	$\mathcal{A}$		a set of A item
case alternatives	A	$::=$	$F(\vec{x}) \to M$
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	$\mathcal{D}, \mathcal{C}, \mathcal{S}$	$::=$	$\mathsf{letrec}\ D\ \mathsf{and}\ \cdots\ \mathsf{and}\ D$

Semantics (small-step reduction)

Client:

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

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

Server:

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

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

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 6. Definition of the Client/Server Machine (CSM).

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 $\longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}}$. This keeps the definition-sets fixed throughout the reduction sequence.

Observation 2. CSM reduction steps can be made in any evaluation context: $(M;\cdot) \longrightarrow (N;\cdot)$ implies $(E[M];\cdot) \longrightarrow (E[N];\cdot)$ and $(E;M) \longrightarrow (E;N)$ implies $(E'[E];M) \longrightarrow (E'[E];N)$. This is a direct consequence of the reduction rules, which are already defined in terms of evaluation contexts.

Lemma 3 (Substitution-Evaluation)**.** *For any substitution σ, we have*

$$(M;\cdot) \longrightarrow_{\mathcal{C},\mathcal{S}} (N;\ \cdot) \ \textit{implies}\ (M\sigma;\ \cdot) \longrightarrow_{\mathcal{C},\mathcal{S}} (N\sigma;\ \cdot)\ \textit{and}$$

$$(E;M) \longrightarrow_{\mathcal{C},\mathcal{S}} (E;N) \ \textit{implies}\ (E;M\sigma) \longrightarrow_{\mathcal{C},\mathcal{S}} (E;N\sigma).$$

$$\boxed{(-)^\circ_- : V_{\lambda_{\text{loc}}} \to V_{\text{CSM}}}$$

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

$$\boxed{(-)^* : M_{\lambda_{\text{loc}}} \to M_{\text{CSM|c}}}$$

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

$$\boxed{(-)^\dagger(-) : (M_{\lambda_{\text{loc}}} \times V_{\text{CSM}}) \to M_{\text{CSM|s}}}$$

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

Figure 7. Term-level translations from λ_{loc} to CSM.

$$\begin{array}{lcl}
\text{coll}_a\ f\ LM & = & f_a(LM) \cup \text{coll}_a\ f\ L \cup \text{coll}_a\ f\ M \\
\text{coll}_a\ f\ \lambda^b x.N & = & f_a(\lambda^b x.N) \cup \text{coll}_b\ f\ N \\
\text{coll}_a\ f\ x & = & f_a(x) \\
\text{coll}_a\ f\ c & = & f_a(c)
\end{array}$$

Figure 8. Generic traversal function for λ_{loc} terms.

Proof. We show that $\mathcal{K} \longrightarrow_{\mathcal{C},\mathcal{S}} \mathcal{K}'$ implies $\mathcal{K}\sigma \longrightarrow_{\mathcal{C},\mathcal{S}} \mathcal{K}'\sigma$. The result for $\longrightarrow\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}}$ then follows by induction on the reduction sequence.

Whether $\mathcal{K}$ is of the form $(M;\cdot)$ or $(E;M)$, decompose M into the form $E'[M']$. The proposition then follows by induction on M'. The proof of each case is a simple matter of pushing the substitutions down through the terms, applying the inductive hypothesis, and pulling them back up the terms. □

Translation from λ_{loc} to CSM

Figures 7–10 give a translation from the client-server λ_{loc} to CSM. Figure 7 gives term-level translations $(-)^\circ$, $(-)^*$ and $(-)^\dagger(-)$, which generate values, client terms, and server terms, respectively. The functions $(-)^*$ and $(-)^\dagger(-)$ are only defined for λ_{loc} client and server terms, respectively. The translation respects this and applies these functions only to terms of the appropriate sort. The translation assumes that the λ_{loc} program being translated is a client term. This is in keeping with the client-server paradigm, where the client initiates the interaction.

The resulting terms make use of function definitions for *apply*, *tramp* and *cont*. The necessary definition sets are produced by the top-level translations, $[\![-]\!]^{\text{c,top}}$ and $[\![-]\!]^{\text{s,top}}$, defined in Figures 9–10, making use of the traversal function coll of Figure 8. Intuitively, the *apply* functions handle all function applications, the *cont* function handles continuation application, and *tramp* is the trampoline, which tunnels server-to-client requests through responses. For this translation, let *arg* and k be special reserved variable names not appearing in the source program.

The function coll traverses a term in a location-sensitive way: it computes the union of $f_a(N)$ for each subterm N of M, where a is the location of the context where N appears.

The bodies of the *apply* functions will have a case for each abstraction appearing in the source term. Each location will have a case for both locations' abstractions; for its 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 transformation; it has a case for evaluating the r.h.s. of each server-located application, a case *App* for applying a function to an argument, and a case for *Top* which is the continuation for returning a value to the client. These correspond to the defunctionalizations of the abstractions that appear in the classic CPS transformation. Recall the translation for applications,

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

and recall that we always need a "top-level" continuation $\lambda x.x$ to extract a value from a CPS term; this corresponds to *Top*.

$$\boxed{[\![-]\!]^{\text{c,top}} : M_{\lambda_{\text{loc}}} \to \mathcal{D}_{\text{CSM}}}$$

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

$$\begin{array}{rcl}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\text{c,fun,aux}}_a & = & \{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \to N^*\{\mathit{arg}/x\}\} \\
& & \text{where } \vec{y} = \text{FV}(\lambda x.N)
\end{array}$$

$$\begin{array}{rcl}
[\![\lambda^{\mathsf{s}} x.N]\!]^{\text{c,fun,aux}}_a & = & \\
\{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \to \mathit{tramp}(\mathsf{req}\ \mathit{apply}\ (\ulcorner \lambda x.N \urcorner(\vec{y}),\, \mathit{arg},\, \mathit{Top}()))\} & & \\
& & \text{where } \vec{y} = \text{FV}(\lambda x.N)
\end{array}$$

$$\begin{array}{rcll}
[\![M]\!]^{\text{c,fun,aux}}_a & = & \{\} & \text{when } M \neq \lambda^a x.N \\
[\![M]\!]^{\text{c,fun}} & = & \text{coll}_{\mathsf{c}}\ ([\![-]\!]^{\text{c,fun,aux}}_-)\ M &
\end{array}$$

Figure 9. Top-level translation from λ_{loc} to CSM (client).

$$\boxed{[\![-]\!]^{\text{s,top}} : M_{\lambda_{\text{loc}}} \to \mathcal{D}_{\text{CSM}}}$$

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

$$\begin{array}{rcl}
[\![\lambda^{\mathsf{s}} x.N]\!]^{\text{s,fun,aux}}_a & = & \{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \to (N^\dagger k)\{\mathit{arg}/x\}\} \\
& & \text{where } \vec{y} = \text{FV}(\lambda x.N)
\end{array}$$

$$\begin{array}{rcl}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\text{s,fun,aux}}_a & = & \\
\{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \to \mathit{Call}(\ulcorner \lambda x.N \urcorner(\vec{y}),\, \mathit{arg},\, k)\} & & \\
& & \text{where } \vec{y} = \text{FV}(\lambda x.N)
\end{array}$$

$$\begin{array}{rcll}
[\![M]\!]^{\text{s,fun,aux}}_a & = & \{\} & \text{when } M \neq \lambda^a x.N \\
[\![M]\!]^{\text{s,fun}} & = & \text{coll}_{\mathsf{c}}\ ([\![-]\!]^{\text{s,fun,aux}}_-)\ M &
\end{array}$$

$$\begin{array}{rcl}
[\![LM]\!]^{\text{s,cont,aux}}_{\mathsf{s}} & = & \ulcorner M \urcorner(\vec{y}, k) \to M^\dagger(\mathit{App}(\mathit{arg}, k)) \\
[\![M]\!]^{\text{s,cont}} & = & \text{coll}_{\mathsf{c}}\ ([\![-]\!]^{\text{s,cont,aux}}_-)\ M
\end{array}$$

Figure 10. Top-level translation from λ_{loc} to CSM (server).

Note that the traversals used in $[\![-]\!]^{\mathsf{c},\mathrm{top}}$ and $[\![-]\!]^{\mathsf{s},\mathrm{top}}$ pass to coll an outermost current location of c; this reflects the requirement that the source program is a client term.

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 x in the $Return(x)$ construction; 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.

3.2 Correctness of the translation from λ_{loc} to CSM

As with the defunctionalization formalization, we again need to compare definition-sets. This time we define the containment relation more generally: it holds just when the names defined in the r.h.s. are all defined in the l.h.s. and upon inspecting corresponding function definitions, either the bodies are identical or they are both case analyses where the l.h.s. contains all the alternatives of the r.h.s.

Definition 3 (Definition containment)**.** We say a definition set $\mathcal{D}$ *contains* $\mathcal{D}'$, written $\mathcal{D} \geqslant \mathcal{D}'$, if 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 $\mathrm{cases}(f, \mathcal{D}) \supseteq \mathrm{cases}(f, \mathcal{D}')$.

Figure 11 defines a reverse translation for the $\lambda_{\text{loc}} \to$ CSM translation. This translation allows us to retract results from CSM into λ_{loc}. All of the functions used in this translation are parameterized on the definition sets $\mathcal{C}$ and $\mathcal{S}$. To extract the alternatives of case-expressions from function bodies, we use a function cases, defined as follows:

Definition 4. Define $\mathrm{cases}(f, \mathcal{D}) = \mathcal{A}$ when

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

Note we are relying 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.

Lemma 4 (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) $(M^\circ)^\bullet_{\mathcal{C},\mathcal{S}} = M$*,*
(ii) $(M^*)^\star_{\mathcal{C},\mathcal{S}} = M$ *and*
(iii) for each K *in CSM:* $(M^\dagger K)^\ddagger_{\mathcal{C},\mathcal{S}} = K^\$ M_{\mathcal{C},\mathcal{S}}$

for those terms on which the relevant operation (respectively $(-)^\circ$*,* $(-)^*$*, or* $(-)^\dagger(-)$*) is defined.*

Proof. By induction on M.

We omit the $(\mathcal{C}, \mathcal{S})$ argument to each of the reverse-translation functions, since it never changes.

- Case x for (i). Trivial.
- Case c for (i). Trivial.
- Case $\lambda^a x.N$ for (i).
  Let x' be a fresh variable.

$$\boxed{(-)^\bullet_{-,-} : V_{\text{CSM}} \to \mathcal{D}_{\text{CSM}} \to \mathcal{D}_{\text{CSM}} \to V_{\lambda_{\text{loc}}}}$$

$$\begin{aligned}
c^\bullet_{\mathcal{C},\mathcal{S}} &= c \\
x^\bullet_{\mathcal{C},\mathcal{S}} &= x \\
(F(\vec{V}))^\bullet_{\mathcal{C},\mathcal{S}} &= \lambda^{\mathsf{c}} x.(N\{x/arg\})^\star_{\mathcal{C},\mathcal{S}}\{\vec{V}^\bullet_{\mathcal{C},\mathcal{S}}/\vec{y}\} \\
&\text{when } (F(\vec{y}) \to N) \in \mathrm{cases}(apply, \mathcal{C}) \text{ and } N \neq tramp(\mathsf{req}\ \cdot\ \cdot) \\
&\text{where } x \text{ fresh} \\
(F(\vec{V}))^\bullet_{\mathcal{C},\mathcal{S}} &= \lambda^{\mathsf{s}} x.(N\{x/arg\})^\ddagger_{\mathcal{C},\mathcal{S}}\{\vec{V}^\bullet_{\mathcal{C},\mathcal{S}}/\vec{y}\} \\
&\text{when } (F(\vec{y}) \to N) \in \mathrm{cases}(apply, \mathcal{S}) \text{ and } N \neq Call(\cdot,\cdot,\cdot) \\
&\text{where } x \text{ fresh}
\end{aligned}$$

$$\boxed{(-)^\star_{-,-} : M_{\text{CSM}|\mathsf{c}} \to \mathcal{D}_{\text{CSM}} \to \mathcal{D}_{\text{CSM}} \to M_{\lambda_{\text{loc}}}}$$

$$\begin{aligned}
V^\star_{\mathcal{C},\mathcal{S}} &= V^\bullet_{\mathcal{C},\mathcal{S}} \\
x^\star_{\mathcal{C},\mathcal{S}} &= x \\
(apply(L, M))^\star_{\mathcal{C},\mathcal{S}} &= L^\star_{\mathcal{C},\mathcal{S}} M^\star_{\mathcal{C},\mathcal{S}} \\
(tramp(\mathsf{req}\ cont\ (K, M)))^\star_{\mathcal{C},\mathcal{S}} &= K^\$_{\mathcal{C},\mathcal{S}}(M^\star_{\mathcal{C},\mathcal{S}}) \\
(tramp(\mathsf{req}\ apply\ (V, W, K)))^\star_{\mathcal{C},\mathcal{S}} & \\
&= (apply(V, W, K))^\ddagger_{\mathcal{C},\mathcal{S}}
\end{aligned}$$

$$\boxed{(-)^\ddagger_{-,-} : M_{\text{CSM}|\mathsf{s}} \to \mathcal{D}_{\text{CSM}} \to \mathcal{D}_{\text{CSM}} \to M_{\lambda_{\text{loc}}}}$$

$$\begin{aligned}
(cont(K, V))^\ddagger_{\mathcal{C},\mathcal{S}} &= K^\$_{\mathcal{C},\mathcal{S}}(V^\bullet_{\mathcal{C},\mathcal{S}}) \\
(apply(V, W, K))^\ddagger_{\mathcal{C},\mathcal{S}} &= K^\$_{\mathcal{C},\mathcal{S}}(V^\bullet_{\mathcal{C},\mathcal{S}} W^\bullet_{\mathcal{C},\mathcal{S}}) \\
(Call(V, W, K))^\ddagger_{\mathcal{C},\mathcal{S}} &= K^\$_{\mathcal{C},\mathcal{S}}(V^\bullet_{\mathcal{C},\mathcal{S}} W^\bullet_{\mathcal{C},\mathcal{S}})
\end{aligned}$$

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

$$\begin{aligned}
k^\$_{\mathcal{C},\mathcal{S}} N &= N \\
(Top())^\$_{\mathcal{C},\mathcal{S}} N &= N \\
(App(V, K))^\$_{\mathcal{C},\mathcal{S}} N &= K^\$_{\mathcal{C},\mathcal{S}}(V^\bullet_{\mathcal{C},\mathcal{S}} N) \\
(F(\vec{V}, K))^\$_{\mathcal{C},\mathcal{S}} N &= \\
&(K^\$_{\mathcal{C},\mathcal{S}}(M^\ddagger_{\mathcal{C},\mathcal{S}}\{\vec{V}^\bullet_{\mathcal{C},\mathcal{S}}/\vec{y}\}))\{N/arg\} \\
&\text{when } (F(\vec{y}) \to M) \in \mathrm{cases}(cont, \mathcal{S}) \\
&\text{and } F \neq Top,\ F \neq App
\end{aligned}$$

Figure 11. Reverse translation from CSM to λ_{loc}.

Take cases on a.
Case $a = \mathsf{s}$:

$$\begin{aligned}
(\lambda^{\mathsf{s}} x.N)^{\circ\bullet} &= (\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}))^\bullet \\
&\qquad \vec{y} = \mathrm{FV}(\lambda x.N) \\
&= \lambda x'.((N^\dagger k)\{arg/x\}\{x'/arg\})^\ddagger \\
&= \lambda x'.((N^\dagger k)\{x'/x\})^\ddagger \\
&= \lambda x.(N^\dagger k)^\ddagger \\
&= \qquad \text{(ind. hyp.)} \\
&\quad \lambda x.N
\end{aligned}$$

huzzah!

Case $a = \mathsf{c}$:

$$\begin{array}{rcll}(\lambda^{\mathsf{c}}x.N)^{\circ\bullet} & = & (\ulcorner\lambda^{\mathsf{c}}x.N\urcorner(\vec{y}))^{\bullet} & \\ & & & \vec{y} = \mathrm{FV}(\lambda x.N) \\ & = & \lambda x'.(N^*\{arg/x\}\{x'/arg\})^{\star} & \\ & = & \lambda x.N^{\star\star} & \\ & = & & \text{(ind. hyp.)} \\ & & \lambda x.N & \end{array}$$

huzzah!

- Cases V for $(-)^*$.

$$\begin{array}{rcll}V^{**} & = & V^{\circ\star} & \\ & = & V^{\circ\bullet} & \\ & = & & \text{(ind. hyp.)} \\ & & V & \end{array}$$

huzzah!

- Cases V for (ii).

$$\begin{array}{rcll}(V^{\dagger}K)^{\ddagger} & = & (cont(K, V^{\circ}))^{\ddagger} & \\ & = & K^{\$}(V^{\circ\bullet}) & \\ & = & & \text{(ind. hyp.)} \\ & & K^{\$}V & \end{array}$$

huzzah!

- Case LM for (ii).

$$\begin{array}{rcll}((LM)^*)^{\star} & = & (apply(L^*, M^*)))^{\star} & \\ & = & L^{\star\star}M^{\star\star} & \\ & = & & \text{(ind. hyp.)} \\ & & LM & \end{array}$$

huzzah!

- Case LM for (iii).

$$\begin{array}{rcll}((LM)^{\dagger}K)^{\ddagger} & = & (L^{\dagger}(\ulcorner M\urcorner(\vec{y}, K)))^{\ddagger} & \vec{y} = \mathrm{FV}(M) \\ & = & & \text{(ind. hyp.)} \\ & & (\ulcorner M\urcorner(\vec{y}, K))^{\$}(L) & \\ & = & (K^{\$}((M^{\dagger}(App(arg, k)))^{\ddagger}))\{L/arg\} & \\ & = & & \text{(ind. hyp.)} \\ & & (K^{\$}((App(arg, k))^{\$}(M)))\{L/arg\} & \\ & = & (K^{\$}(arg\ M))\{L/arg\} & \\ & = & K^{\$}(LM) & \end{array}$$

huzzah!

□

Lemma 5 (Substitution-$(-)^{\bullet}$)**.** *If* $\mathrm{cases}(apply, \mathcal{C})$ *and* $\mathrm{cases}(apply, \mathcal{S})$ *have a case for every constructor appearing in* V *and* W *then*

$$V^{\bullet}_{\mathcal{C},\mathcal{S}}\{W^{\bullet}_{\mathcal{C},\mathcal{S}}/x\} = (V\{W/x\})^{\bullet}_{\mathcal{C},\mathcal{S}}.$$

Proof. By induction on M.

- Case c, x. Trivial.
- Case $F(\vec{V})$.
  Recall that $(F(\vec{V}))^{\bullet}_{\mathcal{C},\mathcal{S}} = (\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\}$.

$$\begin{array}{rcll} & & (\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\}\{W^{\bullet}_{\mathcal{C},\mathcal{S}}/x\} & \\ & = & (\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathcal{C},\mathcal{S}}\{W^{\bullet}_{\mathcal{C},\mathcal{S}}/x\}/\vec{y}\} & \\ & = & & \text{(ind. hyp.)} \\ & & (\lambda^a x.N)\{(\vec{V}\{W/x\})^{\bullet}_{\mathcal{C},\mathcal{S}}/\vec{y}\} & \\ & = & (F(\vec{V}\{W/x\}))^{\bullet}_{\mathcal{C},\mathcal{S}} & \\ & = & (F(\vec{V})\{W/x\})^{\bullet}_{\mathcal{C},\mathcal{S}} & \end{array}$$

huzzah!

□

Lemma 6 (Simulation)**.** *For any term* M *and substitution* σ *in* λ_{loc}*, 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 implications:*

$$\begin{array}{rll}(i) & \textit{if} & M^*\sigma \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} V \\ & \textit{then} & M\sigma^{\bullet}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{c}} V^{\bullet}_{\mathcal{C},\mathcal{S}} \textit{ and} \\ (ii) & \textit{if} & tramp([\ ]); (M^{\dagger}k)\sigma \\ & & \quad \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{C},\mathcal{S}} tramp([\ ]); cont(k, V) \\ & \textit{then} & M\sigma^{\bullet}_{\mathcal{C},\mathcal{S}} \Downarrow_{\mathsf{s}} V^{\bullet}_{\mathcal{C},\mathcal{S}}. \end{array}$$

Proof. By induction on the length of the CSM reduction sequence. Throughout the induction σ is kept general.

Throughout this proof, we make free use of Observation 2 and Lemma 3, showing (respectively) that we can place a reduction inside an evaluation context to get another reduction, and that we can apply a substitution to a reduction to get another reduction.

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

Take cases on the structure of M and split the conclusion into cases for (i) and (ii); in the case of an application LM, take cases on whether L reduces to a client abstraction $\lambda^{\mathsf{c}}x.N$ or to a server abstraction $\lambda^{s}x.N$:

- Case LM for (i) where L reduces to a client abstraction.
  By hypothesis, we have $(LM)^*\sigma \longrightarrow\!\!\!\!\!\rightarrow V$. Recall $(LM)^*\sigma = apply(L^*\sigma, M^*\sigma)$. It must be that $M^*\sigma$ reduces to a value; call it W. It must be that $L^*\sigma$ reduces to a value of the form $F(\vec{y})$. Let N be such that $(F(\vec{y}) \to N^*\{arg/x\}) \in \mathrm{cases}(apply, \mathcal{C})$. (We know it has this form because the cases of *apply* are in the image of the $[\![-]\!]^{\mathsf{c},\mathrm{fun}}$ translation.) Let x be a fresh variable. The reduction follows:

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

  We have $N^*\{\vec{V}/\vec{y}, W/x\} = N^*\{\vec{V}/\vec{y}\}\{W/x\}$ due to the freshness of x.
  Applying the inductive hypothesis three times, we get

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

  The judgment $(LM)\sigma^{\bullet} \Downarrow_{\mathsf{c}} V^{\bullet}$ follows by BETA.
- Case LM for (i) where L reduces to a server abstraction.
  By hypothesis, we have $(LM)^*\sigma \longrightarrow\!\!\!\!\!\rightarrow V$. Recall $(LM)^*\sigma = apply(L^*\sigma, M^*\sigma)$. It must be that $M^*\sigma$ reduces to a value; call

it W. It must be that $L^*\sigma$ reduces to a value of the form $F(\vec{y})$. Let N be such that

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

Let x be a fresh variable. The reduction follows:

$$\begin{array}{rcl}
(LM)^*\sigma & = & apply(L^*\sigma, M^*\sigma) \\
& \longrightarrow\!\!\!\!\!\rightarrow & apply(F(\vec{V}), W) \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp(\mathsf{req}\ apply; (F(\vec{V}), W, Top())) \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ apply(F(\vec{V}), W, Top()) \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ (N^\dagger(Top()))\{\vec{V}/\vec{y}, W/x\} \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ cont(Top(), V) \\
& \longrightarrow\!\!\!\!\!\rightarrow & V
\end{array}$$

Applying the inductive hypothesis three times, we get

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

Also $N\{\vec{V}^\bullet/\vec{y}, W^\bullet/x\} = N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\}$ because of the freshness of x.
The judgment $(LM)\sigma^\bullet \Downarrow_{\mathsf{c}} V^\bullet$ follows by BETA. *huzzah!*

- Case LM for (ii) where L reduces to a server abstraction.
By hypothesis, we have

$$\begin{array}{c}
tramp([\ ]);\ L^\dagger(\ulcorner M \urcorner(\vec{z}, k))\sigma \\
\longrightarrow\!\!\!\!\!\rightarrow tramp([\ ]);\ cont(K, V).
\end{array}$$

Recall that $(LM)^\dagger k = L^\dagger(\ulcorner M \urcorner(\vec{z}, k))$, letting $\vec{z} = \mathrm{FV}(M)$.
It must be that $(M^\dagger k)\sigma$ reduces to a term of the form $cont(k, W)$.
It must be that $(L^\dagger k)\sigma$ reduces to a term of the form $cont(k, F(\vec{y}))$.
Let N be such that $(F(\vec{y}) \to (N^\dagger k)\{arg/x\})) \in \mathrm{cases}(apply, \mathcal{S})$.
Let x be a fresh variable. The reduction follows:

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

Also, because x is fresh, we have

$$N\{\vec{V}^\bullet/\vec{y}, W^\bullet/x\} = N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\}.$$

Applying the inductive hypothesis three times, we get

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

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

- Case LM for (ii) where L reduces to a client abstraction.
By hypothesis, we have

$$\begin{array}{c}
tramp([\ ]);\ (L^\dagger(\ulcorner M \urcorner(\vec{z}, k)))\sigma \\
\longrightarrow\!\!\!\!\!\rightarrow tramp([\ ]);\ cont(k, V).
\end{array}$$

Recall that $(LM)^\dagger k = L^\dagger(\ulcorner M \urcorner(\vec{z}, k))$, letting $\vec{z} = \mathrm{FV}(M)$.
It must be that $(M^\dagger k)\sigma$ reduces to a term of the form $cont(k, W)$.
It must be that $(L^\dagger k)\sigma$ reduces to a term of the form $cont(k, F(\vec{y}))$.
Let N be such that $(F(\vec{y}) \to N^*\{arg/x\}) \in \mathrm{cases}(apply, \mathcal{C})$ and $\vec{y}$ be the variables s.t. $F(\vec{y}) \to N^*\{arg/x\}$ in the client-side def. of $apply$. Also we have $(F(\vec{y}) \to Call(F(\vec{y}), arg, k)) \in \mathrm{cases}(apply, \mathcal{S})$. Let x be a fresh variable. The reduction follows:

$$\begin{array}{rcl}
& & ((LM)^\dagger k)\sigma \\
& = & tramp([\ ]);\ (L^\dagger(\ulcorner M \urcorner(\vec{z}, k)))\sigma \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ cont(\ulcorner M \urcorner(\vec{z}\sigma, k), F(\vec{V})) \\
& \longrightarrow & tramp([\ ]);\ (M^\dagger(App(arg, k))) \\
& & \qquad \{\vec{z}\sigma/\vec{z}, F(\vec{V})/arg\} \\
& = & tramp([\ ]);\ (M^\dagger k)\{\sigma,\ App(F(\vec{V}), k)/k\} \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ cont(App(F(\vec{V}), k), W) \\
& \longrightarrow & tramp([\ ]);\ apply(F(\vec{V}), W, k) \\
& \longrightarrow & tramp([\ ]);\ Call(F(\vec{V}), W, k) \\
& \longrightarrow & tramp(Call(F(\vec{V}), W, k)) \\
& \longrightarrow & tramp(\mathsf{req}\ cont\ (k, apply(F(\vec{V}), W))) \\
& \longrightarrow & tramp(\mathsf{req}\ cont\ (k, N^*\{\vec{V}/\vec{y}, W/x\})) \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp(\mathsf{req}\ cont\ (k, V)) \\
& \longrightarrow\!\!\!\!\!\rightarrow & tramp([\ ]);\ cont(k, V)
\end{array}$$

Because of the freshness of x, we have $N\{\vec{V}^\bullet/\vec{y}, W^\bullet/x\} = N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\}$.
Applying the inductive hypothesis three times, we get

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

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

- Case V. Using the substitution lemma (Lemma 5) and the inverseness of $(-)^\bullet$ to $(-)^\circ$, we get $(V^\circ\sigma)^\bullet = V\sigma^\bullet$.
In both cases, for server and client, the reduction is of zero steps:
For client, $V^*\sigma \longrightarrow\!\!\!\!\!\rightarrow V^\circ\sigma$. The judgment $V\sigma^\bullet \Downarrow_{\mathsf{c}} V\sigma^\bullet$ follows by VALUE.
For server,

$$\begin{array}{c}
tramp([\ ]);\ (V^\dagger k)\sigma \\
\longrightarrow\!\!\!\!\!\rightarrow tramp([\ ]);\ cont(k, V^\circ\sigma).
\end{array}$$

The judgment $V\sigma^\bullet \Downarrow_{\mathsf{s}} V\sigma^\bullet$ follows by VALUE.

□

4. A richer calculus

As a hypothetical language feature, we consider detaching location annotations from abstractions, permitting any term to be wrapped in location brackets.

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 (big-step)

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

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

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

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

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

Figure 12. 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 13. Translation from $\lambda_{\langle\rangle}$ to λ_{loc}.

The calculus $\lambda_{\langle\rangle}$ in Figure 12 adds location brackets $\langle\cdot\rangle^a$ to λ_{loc} 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 the term M at location a and returns the value to the location b. Note that unannotated λ-abstractions are not treated as values: all values must be mobile, and yet the interpretation of an unannotated abstraction is that its body will be executed at the location corresponding to the nearest containing annotation. Thus the abstraction itself must get annotated with this location, and the ABSTR rule attaches this annotation when it is not already provided.

Figure 13 gives a translation from $\lambda_{\langle\rangle}$ to λ_{loc}. 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 the nearest containing brackets.

To argue that this translation is correct in the same say as the previous translation, we would need a reverse translation, but there is a problem: the translation is not injective. We could resort to some hack to distinguish the terms, or we could try to prove a looser relationship, perhaps using a simulation relation. We don't provide a correctness proof here, but hope that the given interpretation is enough to persuade the reader that the translation is reasonable.

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 Lambda 5 (Murphy et al. 2004; Murphy 2007) is a small calculus with constructs for controlling the location and movement of terms and values. Lambda 5 offers fine control over the runtime movements of code and data, whereas our calculus uses the simple 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 closure conversion.

Neubauer and Thiemann give an algorithm for splitting a location-annotated sequential program into separate concurrent programs that communicate over channels (Neubauer and Thiemann 2005). 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 that mediator. Neubauer and Thiemann use session types to show that the various processes' use of channels are type-correct over the course of the interaction. Such a type analysis could be profitably applied to λ_{loc}.

For security purposes, Zdancewic, Grossman and Morrissett developed a calculus with brackets (Zdancewic et al. 1999), 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.

Defunctionalization After first being introduced in a lucid but informal account by John Reynolds (Reynolds 1972), defunctionalization has been formalized and verified in a typed setting in several papers (Bell and Hook 1994; Bell et al. 1997; Pottier and Gauthier 2004; Nielsen 2000; Banerjee et al. 2001). These formalizations are complicated by the types; we trade type safety for economy of presentation. Danvy and Nielsen (2001) explore a number of uses and interesting relationships provided by defunctionalization.

Defunctionalization is very similar to lambda-lifting (Johnsson 1985), the essential difference being that lambda-lifting doesn't reify a closure as a denotable value. Thus it would not be applicable here, where we need to serialize the function to send across the wire.

As noted in the introduction, Murphy (2007) 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 a system.

Continuation-Passing The continuation-passing transformation has a long and storied history, going back to the 1970s (Fischer 1972; Plotkin 1975). Our treatment owes much to the presentation and results of "A Reflection on Call-By-Value" (Sabry and Wadler 1997).

6. Conclusions and Future Work

We've shown how to compile a located λ-calculus to an asymmetrical, stateless client-server machine by using a CPS-transformation and "trampoline" effectively to represent the server's call stack as a value on the client. In the future, we hope to extend the source calculus by adding features such as 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 "richer calculus" with location brackets in the Links language.

The present work begins with a source calculus with location annotations, but the activity of annotation may burden the program-

mer. Considering that some resources are available only at some locations, it should be possible to automatically assign location annotations so as to reduce communication costs, rather than requiring the programmer to carefully annotate the program. 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 and would be interesting to explore.

References

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.

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

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

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, 2006.

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

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

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. ISBN 3-387-15975-4.

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

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.

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.

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

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

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

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. doi: http://doi.acm.org/10.1145/800194.805852.

Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

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.