A Symmetric Modal Lambda Calculus for Distributed Computing *

Tom Murphy VII
Carnegie Mellon
tom7@cs.cmu.edu

Karl Crary
Carnegie Mellon
crary@cs.cmu.edu

Robert Harper
Carnegie Mellon
rwh@cs.cmu.edu

Frank Pfenning
Carnegie Mellon
fp@cs.cmu.edu

Abstract

We present a foundational language for spatially distributed programming, called Lambda 5, that addresses both mobility of code and locality of resources.

In order to construct our system, we appeal to the powerful propositions-as-types *interpretation of logic. Specifically, we take the* possible worlds *of the intuitionistic modal logic IS5 to be nodes on a network, and the connectives* $\Box$ *and* $\Diamond$ *to reflect mobility and locality, respectively.*

We formulate a novel system of natural deduction for IS5, decomposing the introduction and elimination rules for $\Box$ *and* $\Diamond$*, thereby allowing the corresponding programs to be more direct. We then give an operational semantics to our calculus that is type-safe, logically faithful, and computationally realistic.*

1 Introduction

The popularity of the Internet has enabled the possibility of large-scale distributed computation. Distributed programming is especially popular for scientific computing tasks. The goal of this paper is to present a foundational programming language for spatially distributed computing. Scientific computing tasks often require the physical distribution of computational resources and sensing instruments. Therefore, to be relevant, our language must address both the mobility of code and the locality of fixed resources.

Due to aesthetic considerations, we wish to take a *propositions-as-types* interpretation of an appropriate logic to form the basis of our programming language. Moreover, since the type systems of realistic languages such as ML and Haskell come from the same source, our constructs will smoothly integrate with such languages. To make use of this interpretation, our requirements are as follows. First, we must be able to give a realistic operational semantics to our system, since we want it to be relevant to real programming languages. Second, the corresponding logic must be well-behaved; it must be locally sound and complete, and equivalent to an appropriate sequent calculus. Because of its ability to represent spatial reasoning, we argue that intuitionistic modal logic forms an excellent basis for distributed computing. Our modal logic, called Lambda 5, has both a realistic operational semantics and a well behaved proof theory.

Just as propositional logic is concerned with *truth*, modal logic is concerned with truth relative to different *worlds*. The worlds are related by an *accessibility relation* whose properties distinguish different modal logics. We will explain our choice of accessibility relation below.

Modal logic is generally concerned with two forms of propositions: $\Box A$, meaning that A is true *in every (accessible) world*, and $\Diamond A$, meaning that A is true *in some (accessible) world*. Our computational interpretation realizes these worlds as the nodes in a network. Because our model is a computer network where all nodes can communicate with each other equally, we choose an accessibility relation that is reflexive, symmetric, and transitive, which leads to the intuitionistic modal logic IS5 [14]. A value of type $\Box A$ represents mobile code of type A that can be executed at any world; a value of type $\Diamond A$ represents the address of a remote value of type A. To illustrate our interpretation, we present some characteristic true propositions in IS5 and their intuitive justifications.

$\Box \mathbf{A} \supset \mathbf{A}$ – Mobile code can be executed.
$\Box \mathbf{A} \supset \Box\Box \mathbf{A}$ – Mobile code is itself mobile.
$\mathbf{A} \supset \Diamond \mathbf{A}$ – We can create an address for any value.
$\Diamond\Diamond \mathbf{A} \supset \Diamond \mathbf{A}$ – We can obtain a remote address.
$\Diamond \mathbf{A} \supset \Box\Diamond \mathbf{A}$ – Addresses are mobile values.
$\Diamond\Box \mathbf{A} \supset \Box \mathbf{A}$ – We can obtain a remote mobile value.

The last two provable propositions are especially relevant, and are only true because our accessibility relation is symmetric. These theorems are actually some standard axioms for a Hilbert-style presentation of IS5. We opt for a judgmental presentation, so all of these are provable propositions in Lambda 5. In section 4.1 we look at the actual proof terms for some of these sentences and their computational content.

*The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination."

On the other hand, the following are not provable:

$\nvdash$ $\mathbf{A} \supset \Box\mathbf{A}$ – Not all local values are mobile.

$\nvdash$ $\Diamond\mathbf{A} \supset \mathbf{A}$ – We cannot obtain all remote values.

Simpson, in his Ph.D. thesis [17], provides an account of intuitionistic modal logic based on a generic multiple-world semantics. Two aspects prevent us from using his formulation directly. First, his system is generalized to support accessibility relations that are arbitrary geometric theories. For our use of IS5, there is no relevant computational content to a proof that two worlds are related. We therefore dispense with judgments of the accessibility relation (as Kanger [7]) and simply collect a list of worlds that are mutually interaccessible.

The second issue requires a more significant change. Simpson's rules act non-locally in the sense that they often use assumptions from one world to conclude facts in another world. This leads to proof terms that are inefficient at best, and at worst do not even fit our computational model. (In section 4.4 we make this comparison concrete.) Our solution here is to decompose the rules for the $\Box$ and $\Diamond$ connectives into restricted rules that act locally, and motion rules which extend our reasoning across world boundaries. In doing so we nonetheless preserve the duality of the connectives and the desirable logical qualities, as demonstrated in section 3.

This work focuses on *spatially* distributed computing. Many distributed applications are also concurrent, but we deliberately do not address concurrency in order to more clearly isolate and explain spatial distribution in a foundational way. We believe that adding concurrency to the language poses no special issues, and expect to integrate it in an implementation of a Lambda 5-based programming language as future work.

The remainder of the paper proceeds as follows. We begin the first half by presenting our logic in judgmental style and proving standard properties about it. We then present a sequent calculus based on Simpson's IS5 which admits cut and is equivalent to our system of natural deduction. This yields a normal form theorem for our system of natural deduction, validating its design. In the second half of the paper we present the operational semantics of Lambda 5 based on a network abstraction. For this semantics we show type safety and present several examples. We conclude with a discussion of related work and plans for the future.

This paper has a companion technical report [10] with most proofs in full detail. The relationship between natural deduction and sequent formulations of IS5, as well as the admissibility of cut and the normalization theorem have been mechanized in the Twelf system [13] and verified using its metatheorem checker [16].[1]

[1]They can be found at `http://www.cs.cmu.edu/~concert/`.

2 Judgmental Lambda 5

Recall that our logic expresses truth relative to worlds. Following Martin-Löf [8], we employ the notion of a *hypothetical judgment*, which is an assertion of judgment under certain assumptions. The judgments that capture our notion of truth *at a particular world* have the form

$$\Omega; \Gamma \vdash A \,\mathsf{true} \,@\, \omega$$

This judgment expresses that under the assumptions in Γ and Ω, the proposition A is true at the world ω. Γ is a set of assumptions of the form $x_i : A_i \,\mathsf{true} \,@\, \omega_i$ where all variables x_i are distinct. Reasoning about truth at worlds requires reasoning about worlds. For S5, the only thing we need to know about a world is that it exists, so Ω is a set of assumptions of the form ω_i exists where all variables ω_i must be distinct. However, we elide "true" and "exists" when writing judgments for brevity. We only consider judgments that are well-formed in the following sense: All world variables that appear attached to assumptions or in the conclusion are present in Ω.[2]

We define the meaning of our logical connectives by way of introduction (marked I) and elimination (marked E) rules. Introduction rules state the conditions under which a formula involving the connective is true. Elimination rules state how we can use a formula involving the connective whose truth we know. As discussed earlier, we have in addition special rules that encapsulate the mobility of certain connectives, which also contribute to the definition of their meaning.

We consider only implication ($\supset$), necessity ($\Box$) and possibility ($\Diamond$). As discussed in section 5, conjunction and truth are easy to support, while disjunction and falsehood require further consideration for a satisfactory operational semantics.

The entire natural deduction system is given in figure 1. These rules include proof terms, which will be necessary for the operational semantics (section 4). They can be ignored for the present discussion.

The hypothesis rule and rules for implication are standard. They act locally in the sense that the world ω remains the same everywhere.

In order to prove that a proposition is true everywhere, we prove its truth at a hypothetical world where nothing is known but its existence. This explains the $\Box$ introduction rule. The $\Box$ elimination rule states that if $\Box A$ is true here (meaning A is true everywhere) then A is true here. Note that $\Box E$ is different from Simpson's corresponding rule and only strong enough in conjunction with the *fetch* rule explained below.

[2]We could ensure this as a theorem by adding a well-formedness condition on Γ under Ω in the hypothesis rule. To simplify the discussion we take the common shortcut of ruling out ill-formed contexts from the beginning.

$$\frac{\Omega;\Gamma, x : A@\omega \vdash M : A'@\omega}{\Omega;\Gamma \vdash \lambda x.M : A \supset A'@\omega} \supset I \qquad \frac{\begin{array}{c}\Omega;\Gamma \vdash N : A'@\omega \\ \Omega;\Gamma \vdash M : A' \supset A@\omega\end{array}}{\Omega;\Gamma \vdash MN : A@\omega} \supset E \qquad \frac{\omega \in \Omega}{\Omega;\Gamma, x : A@\omega, \Gamma' \vdash x : A@\omega}\ \text{hyp}$$

$$\frac{\omega' \text{ fresh} \quad \Omega, \omega';\Gamma \vdash M : A@\omega' \quad \omega \in \Omega}{\Omega;\Gamma \vdash \texttt{box}\ \omega'.M : \Box A@\omega} \Box I \qquad \frac{\Omega;\Gamma \vdash M : \Box A@\omega}{\Omega;\Gamma \vdash \texttt{unbox}\, M : A@\omega} \Box E \qquad \frac{\omega \in \Omega \quad \Omega;\Gamma \vdash M : \Diamond A@\omega'}{\Omega;\Gamma \vdash \texttt{get}\ \langle\omega'\rangle M : \Diamond A@\omega}\ \text{get}$$

$$\frac{\begin{array}{c}\omega' \text{ fresh} \quad \Omega;\Gamma \vdash M : \Diamond A@\omega \\ \Omega, \omega';\Gamma, x : A@\omega' \vdash N : B@\omega\end{array}}{\Omega;\Gamma \vdash \texttt{letd}\, \omega'.x = M\, \texttt{in}\, N : B@\omega} \Diamond E \qquad \frac{\Omega;\Gamma \vdash M : A@\omega}{\Omega;\Gamma \vdash \texttt{here}\, M : \Diamond A@\omega} \Diamond I \qquad \frac{\omega \in \Omega \quad \Omega;\Gamma \vdash M : \Box A@\omega'}{\Omega;\Gamma \vdash \texttt{fetch}[\omega']M : \Box A@\omega}\ \text{fetch}$$

Figure 1. Lambda 5 natural deduction

For $\Diamond$ we have the dual situation. If A holds here, then we know it is true *somewhere*; this is $\Diamond$ introduction. The $\Diamond$ elimination rule states that if we know $\Diamond A$, then we can reason as if A holds at some hypothetical world about which nothing else is known. Both of these rules have unusual restrictions when compared to other systems: in $\Diamond I$ the premise and conclusion are at the same world; in $\Diamond E$ the first and second premise (and therefore also the conclusion) are at the same world.

Finally, we have rules that explicitly represent the mobility of $\Box$ and $\Diamond$ terms. The *fetch* rule states that if $\Box A$ holds at ω, then it holds at another world ω', provided that ω' exists. In other words, if A is true everywhere from the perspective of one world, then it is true everywhere from the perspective of any other world. Similarly, *get* states that if A is true *somewhere* from the perspective of one world, then it is also true somewhere from the perspective of any other existing world.

It's worth noting that *get* and *fetch* are the source of symmetry in Lambda 5. They are what allow us to prove the characteristic S5 axioms $\Diamond\Box A \supset \Box A$ and $\Diamond A \supset \Box\Diamond A$. Operationally, all communication on the network will be encapsulated in these two rules.

Because we have a hypothetical judgment, we expect to have a substitution principle that allows us to "fill in" assumptions with proofs.

Theorem 1 (Substitution)
If $\mathcal{D} :: \Omega;\Gamma \vdash M : A@\omega$
and $\mathcal{E} :: \Omega;\Gamma, x : A@\omega \vdash N : B@\omega'$
then $\mathcal{F} :: \Omega;\Gamma \vdash [M/x]N : B@\omega'$.

Proof is by structural induction on the derivation $\mathcal{E}$, omitted here.

Similarly, because we have assumptions about the existence of worlds, we have a world substitution principle, which is also a theorem of our logic.

Theorem 2 (World Substitution)
If $\omega' \in \Omega$
and $\mathcal{E} :: \Omega, \omega;\Gamma \vdash M : A@\omega''$
then $\mathcal{F} :: [\omega'/\omega](\Omega;\Gamma \vdash M : A@\omega'')$

Here we mean the substitution to apply to the entire judgment, particularly the world in the conclusion. Proof is again by structural induction on $\mathcal{E}$, omitted here.

We also have the familiar principles of weakening and contraction, for both world and truth assumptions.

As per our criteria, Lambda 5 natural deduction is locally sound and complete. We omit the proofs for space (they appear in the technical report); moreover, these conditions are weaker than normal because of our motion rules. Local soundness, for instance, ensures that our elimination rules are not too strong—if we introduce a connective and then immediately eliminate it, we can find justification for our conclusion. Because this property speaks only of introduction and elimination rules (which traditionally explain a connective completely), it is unable to tell us anything about the motion rules.

A much stronger condition comes by way of equivalence to an appropriate sequent calculus. Because sequent calculus proofs have a particular form, this gives us immediate theoretical and philosophical results that subsume the local properties above. The following section proves this correspondence and describes some of the results that follow. The operational interpretation (section 4) does not depend on it.

3 Sequent Calculus

We establish a (cut-free) sequent calculus SS5 with the following basic judgment:

$$\Omega;\Gamma \longrightarrow A@\omega$$

This judgment states that with truth assumptions Γ and world assumptions Ω, the proposition A is true at ω. The rules of the sequent calculus SS5 are given in figure 2. Note that this calculus admits non-local reasoning in the $\Box L$ and

$$\frac{\Omega;\Gamma, A\supset B@\omega \longrightarrow A@\omega \quad \Omega;\Gamma, A\supset B@\omega, B@\omega \longrightarrow C@\omega'}{\Omega;\Gamma, A\supset B@\omega \longrightarrow C@\omega'} \supset L \qquad \frac{\Omega;\Gamma, A@\omega \longrightarrow B@\omega}{\Omega;\Gamma \longrightarrow A\supset B@\omega} \supset R \qquad \frac{}{\Omega,\omega;\Gamma, A@\omega \longrightarrow A@\omega}\text{ init}$$

$$\frac{\omega' \text{ fresh} \quad \Omega,\omega';\Gamma, \Diamond A@\omega, A@\omega' \longrightarrow C@\omega''}{\Omega;\Gamma, \Diamond A@\omega \longrightarrow C@\omega''} \Diamond L \qquad \frac{\Omega,\omega;\Gamma \longrightarrow A@\omega'}{\Omega,\omega;\Gamma \longrightarrow \Diamond A@\omega} \Diamond R$$

$$\frac{\Omega,\omega';\Gamma, \Box A@\omega, A@\omega' \longrightarrow C@\omega''}{\Omega,\omega';\Gamma, \Box A@\omega \longrightarrow C@\omega''} \Box L \qquad \frac{\omega' \text{ fresh} \quad \Omega,\omega,\omega';\Gamma \longrightarrow A@\omega'}{\Omega,\omega;\Gamma \longrightarrow \Box A@\omega} \Box R$$

Figure 2. Sequent calculus SS5

$\Diamond R$ rules, and lacks the motion rules from natural deduction. It is a version of Simpson's $\mathbf{L}_{\Box\Diamond}(\mathcal{T})$ specialized to the case of interaccessible worlds (IS5).

The sequent calculus still admits world substitution, which is straightforward and therefore omitted here. It is also immediate to prove that weakening and contraction are admissible rules which do not change the structure of a derivation. The substitution principle for derivations turns into the admissibility of cut, which states that a proof of $A@\omega$ licenses us to use $A@\omega$ as a hypothesis.

Theorem 3 (Admissibility of Cut (SS5))
If $\mathcal{D} :: \Omega;\Gamma \longrightarrow A@\omega$
and $\mathcal{E} :: \Omega;\Gamma, A@\omega \longrightarrow B@\omega'$
then $\mathcal{F} :: \Omega;\Gamma \longrightarrow B@\omega'$.

The proof proceeds by a simple lexicographic induction on (in order) the cut formula A, the derivation $\mathcal{D}$, and the derivation $\mathcal{E}$, following Pfenning [11]. To reduce extraneous $\Box$ and $\Diamond$ formulas we need world substitution. This proof is new[3] and has been verified using the Twelf metatheorem checker. It is presented in full detail in the companion technical report [10].

Each rule in the sequent calculus, when read bottom-up, proceeds by decomposing the principle connective of a proposition of the sequent in the antecedent (by a *left rule*) or the succeedent (by a *right rule*). Unlike natural deduction, a sequent derivation therefore embodies what Martin-Löf calls a *verification*: a canonical proof of a proposition which proceeds only by analysis of the proposition to be proved. This gives us an important orthogonality condition: we can extend or limit our logic to different sets of connectives without affecting the provability of propositions involving those connectives.

It is now a relatively simple matter to validate the correctness of our natural deduction system. First, we have to show that every proposition that has a proof (in natural deduction) has a verification (in the sequent calculus). This is the global analogue of the local soundness property. Second, we have to show that every proposition that has a verification, has a verification where the *init* rule is applied only to an atomic proposition. This is the global analogue of the local completeness property, ensuring that the left rules are strong enough to derive $\Omega,\omega;\Gamma, A@\omega \longrightarrow A@\omega$ by decomposing A all the way to its atomic constituents. We omit the proof of the latter property since it is an entirely straightforward induction on the structure of A.

Theorem 4 (Equivalence of Lambda 5 and SS5)
$\Omega;\Gamma \vdash A@\omega$ *iff* $\Omega;\Gamma \longrightarrow A@\omega$.

Each direction is proved by structural induction on the input derivation. In the Lambda 5 to SS5 direction, we use the cut theorem for SS5. These two proofs have also been fully formalized and checked in Twelf.

We can exploit the computational content of this metatheoretic proof to translate an arbitrary natural deduction to the sequent calculus and then back. Analysis of the proofs of theorem 4 shows that the resulting natural deduction will satisfy a very restricted normal form. This normal form satisfies the subformula property and can be constructed using only introduction rules bottom-up and only elimination rules top-down until an assumption matches the conclusion. Moreover, the *fetch* rule needs to be used only immediately above a $\Box E$ rule. Similarly, the *get* rule needs to be used only immediately before the left premise of a $\Diamond E$ rule or immediately below a $\Diamond I$ rule. Therefore we claim that the decomposition of the introduction and elimination rules into local rules and movement rules has not destroyed the logical reading of deductions.

The sequent calculus makes it easy to see that some propositions are not provable. Working bottom-up, we see that the proposition $A \supset \Box A$ is unprovable after applying $\supset R$ and $\Box R$, and being left with no rules to continue. Similarly, after an application of $\supset R$ and $\Diamond L$, we see that $\Diamond A \supset A$ is also unprovable. Decidability of IS5 is another easy consequence [17].

Having justified Lambda 5 as a logic, we now switch gears to its interpretation as a type system for a distributed

[3] Simpson [17] achieved the same result indirectly via natural deduction

programming language.

4 Operational Interpretation

We can associate a programming language with our logic by viewing propositions as types and proofs of those propositions as programs.

Our operational semantics defines an abstract machine: a network and the steps of computation of a program distributed among its nodes. Because we focus on distributed—as distinguished from concurrent—computation, our abstract machine is sequential and deterministic. The network consists of a fixed number of hosts named $\mathbf{w}_i$. Each world has associated with it some state describing its execution context (explained later) and a table. This table stores mappings from labels ℓ to values. These labels, when paired with the world name, form a portable address that others can use to refer to this value.

Before we describe this machine in detail, we revisit the previously ignored proof terms from figure 1. These proof terms form the external language of Lambda 5. As remarked previously, we give the following computational interpretation to our connectives. As usual, values of type $A \supset B$ are functions from A to B. Values of type $\Box A$ are pieces of quoted code that can be run anywhere to produce a value of type A. A value of $\Diamond A$ takes the form $\mathbf{w}.\ell$—a pair of a world name and label. This is an address of a table entry at **w** containing a value of type A.

The proof term for $\Box I$ is $\texttt{box}\ \omega'.M$. It binds the world variable ω' within M, which must be well-typed at ω'. We do not evaluate under the `box`—doing so is unsound in the presence of effects.[4] Straightforwardly, `unbox` instantiates the hypothetical world with the actual current world and then evaluates the contents of the `box`. The term $\texttt{fetch}[\omega']M$ performs a remote procedure call (RPC), executing the code M at ω' and then retrieving the resulting value, which must have $\Box$ type.

The introduction form for $\Diamond$ is $\texttt{here}\ M$. Operationally, we will evaluate the term M and insert the value in a table at the current world. It will be given a new label, and the address will be $\mathbf{w}.\ell$. The elimination form, $\texttt{letd}\,\omega.x = M\ \texttt{in}\ N$, evaluates M to one of these pairs, and then binds variables for the label and world for the purposes of evaluating N. World-label pairs make sense globally, so we are able to retrieve them with $\texttt{get}\ \langle\omega'\rangle M$, which behaves as `fetch` but returns a value of $\Diamond$ type.

Note that in both RPC forms we must send the term M to the remote host. Though this term has $\Box$ or $\Diamond$ type, it is an arbitrary expression, not yet a `box` or $\mathbf{w}.\ell$. In this sense all code must be "mobile;" however, we are able to distinguish between mobile code that can be transmitted to only one location ($A@\omega$) and code that is universally mobile ($\Box A$).

In order to ground our discussion of the operational machinery, we present in the next section some examples of Lambda 5 programs and their intended behavior.

4.1 Examples

As examples, we revisit several of the axioms informally explained in the introduction.

Let's look again at the symmetry axiom $\Diamond\Box A \supset \Box A$. We consider this our key example, because it encapsulates the notion of moving mobile code from some other location to our location. Here is a Lambda 5 proof term for it:

$$\lambda x.\ \texttt{letd}\,\omega.y = x\ \texttt{in}\ \texttt{fetch}[\omega]\ y$$

This term deconstructs the diamond to learn the world at which the mobile code exists, and then *fetch*es it to the current world.

The axiom $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$ is provable in any intuitionistic modal logic based on a Kripke model, regardless of the accessibility relation.[5] Here is the proof term, assuming that it lives at ω.

$$\begin{array}{l} \lambda f.\ \texttt{box}\ \omega'.\lambda y. \\ \quad \texttt{unbox}(\texttt{fetch}[\omega](f(\texttt{get}\ \langle\omega'\rangle\ \texttt{here}\ y))) \end{array}$$

This proof is a bit surprising. We take f, which lives at ω. The boxed code takes $y : A$, which lives at ω'. We then switch *back* to ω in order to apply f; to do so we *get* a $\Diamond A$ from ω'. This back-and-forth is inevitable because we cannot apply f until $\Diamond A$ is true, and $\Diamond A$ is only true once we begin to prove the boxed conclusion.

Let's take a look at the "shortcut" axiom $\Diamond\Diamond A \supset \Diamond A$.

$$\lambda r.\ \texttt{letd}\,\omega'.x = r\ \texttt{in}\ \texttt{get}\ \langle\omega'\rangle x$$

The program simply follows $\Diamond\Diamond A$ to the place where $\Diamond A$ is true, and retrieves that address with *get*.

The other symmetry axiom $\Diamond A \supset \Box\Diamond A$ has two different proofs that are each interesting. These proof terms are well-typed at ω:

$$\begin{array}{ll} 1. & \lambda x.\ \texttt{letd}\,\omega'.y = x \\ & \quad \texttt{in}\ \texttt{box}\ \omega''.\ \texttt{get}\ \langle\omega'\rangle(\texttt{here}\ y) \\ 2. & \lambda x.\ \texttt{box}\ \omega'.\ \texttt{get}\ \langle\omega\rangle\ x \end{array}$$

In the first proof, we deconstruct the diamond and republish it at ω' each time the box is opened. This keeps ω out of the loop at the expense of redundant table entries. In the

[4] The `here` construct is effectful, because it modifies the local table, and we also want our language to scale to traditional effects such as references.

[5] However, it is not provable in some other computational modal logics such as the judgmental S4 due to Pfenning and Davies [12] where necessity is taken to mean provability with *no* assumptions.

second proof, we do not republish the address but simply *get* it from ω.

In section 4.4 we justify our decomposition by comparing some of these proof terms to a hypothetical system where the rules act non-locally.

4.2 Type System

The syntax of our type system and operational semantics is given in figure 3. As mentioned, we give specific names, **w**, to hosts in our network. Because we still have hypothetical worlds ω (for the introduction of $\Box$ or elimination of $\Diamond$), we have world expressions (written as a Roman w) which range over both ω and **w**.

The class of expressions is the same as proof terms in our logic except for the appearance of labels ℓ. We have seen labels as a component of an address of type $\Diamond A$. These values of diamond type are well-typed at any world. In comparison, "disembodied" labels ℓ are well-typed only in the world where their table lives. For example, suppose there is a resource of type A in the table at world $\mathbf{w}_1$. If the label ℓ refers to that resource, then it will have type $A@\mathbf{w}_1$. On the other hand, the address $\mathbf{w}_1.\ell$ can have type $\Diamond A@\mathbf{w}_2$—at a different world.

As a result, a term that is physically present at one node may nonetheless contain components that are only well typed at other worlds. One consequence of our safety theorem is that these subterms will only be evaluated in the appropriate worlds!

The tables at each world (b) are just mappings from labels to values. The type of these tables is τ, a mapping from labels to types.

Our abstract machine is continuation based. For instance, an attempt to evaluate an application MN will result in a $\circ\ N$ frame being pushed onto the continuation. This continuation expects a lambda value, at which point it will begin evaluating N. New in our system is the idea that continuations can span multiple worlds. This arises from the RPC mechanisms. For instance, suppose we evaluate `fetch`$[\mathbf{w}']M$ at **w**. To do so, we suspend our current work at **w** and begin a new continuation on $\mathbf{w}'$ to evaluate M. The bottom of this continuation will be `return` **w**, which awaits a value to return to our old continuation at **w**.

Because RPCs can be reentrant in the sense that code we invoke in one world may in turn invoke code back in the original world, we may have multiple outstanding continuations. However, because the computation is serial, a stack of pending continuations suffices. So, a continuation k is a stack of frames f with either `return` **w** or `finish` at its bottom. A continuation stack C is simply a list of pending continuations. `finish` is the very bottom of the entire network-wide continuation, and when reached represents the final answer of our program.

Now we can discuss network configurations. A configuration $\mathbb{W}$ is a mapping from world constants to their current continuation stacks and tables. The configuration changes as a program is executed; the continuation stacks grow and shrink, and the table monotonically accumulates new values. However, the domain of $\mathbb{W}$ remains constant.

A network state $\mathbb{N}$ is a configuration paired with a cursor. The cursor is of the form $\mathbf{w} : [k, M]$ and represents the current focus of computation. The expression M is currently pending evaluation, the continuation k is the currently active continuation, and the world **w** is where the computation is taking place. The world **w** must of course be in the configuration, but the continuation k does *not* appear in that world's continuation stack.

The final point of the syntax is the configuration type Σ. This simply describes the "type" of the network by mapping world constants to table types.

The natural deduction system given in section 2, with proof terms, can be thought of as the type system for the *external language* of Lambda 5 programs. However, we must extend this type system to talk about networks, tables, and continuations in order to state properties about our abstract machine. To do this, we need a number of new judgments.

The typing judgment $\Sigma; \Omega; \Gamma \vdash M : A@\mathrm{w}$ simply extends the natural deduction judgment to incorporate config types and world expressions. The definition of the well-formedness condition $\Sigma; \Omega \vdash \mathrm{w}$ is omitted for space. It is straightforward: world variables are well-formed when they are in Ω and world names are well-formed when they are in the domain of the configuration type Σ. We also omit the definition of $\Sigma \vdash \ell : A@\mathbf{w}$, which simply ensures that **w**'s entry in Σ maps ℓ to A. The last omitted definition is of table well-formedness, $\Sigma \vdash b@\mathbf{w}$. A table is well-formed when it contains exactly the same labels as its table type claims, and each of the values has the correct type under Σ. We will define the continuation typing judgment $\Sigma \vdash \mathbb{W}; k : A@\mathbf{w}$, which says that the continuation k (and configuration $\mathbb{W}$) expects values of type A at world **w**.

All of these judgments are used to conclude well-formedness for an entire network state, which is written $\Sigma \vdash \mathbb{N}$. The type system reuses the rules from Lambda 5 natural deduction (figure 1) with the following changes. First, we systematically change each judgment of $\Omega; \Gamma \vdash M : A@\omega$ to $\Sigma; \Omega; \Gamma \vdash M : A@\mathrm{w}$, except in the $\Box I$ rule, where the premise must still be concluded at the new hypothetical world ω'. Second, world existence conditions $\omega \in \Omega$ are replaced by the world expression well-formedness condition $\Sigma; \Omega \vdash \mathrm{w}$. Finally, we add a number of new rules from figures 4 and 5, including new typing rules for $\mathbf{w}.\ell$ and disembodied ℓ, called *dia* and *lab*.

Typing of continuations is fairly straightforward. Recall that the judgment records the type *expected* by the continuation, not the type it produces. The most interesting rule

$$
\begin{array}{rcll}
\text{types} & A,B & ::= & \Box A \mid A \supset B \mid \Diamond B \\
\text{configs} & \mathbb{W} & ::= & \{\mathbf{w}_1 : \langle C_1, b_1\rangle, \cdots\} \\
\text{networks} & \mathbb{N} & ::= & \mathbb{W}; \mathbf{w} : [k, M] \\
\text{tables} & b & ::= & \bullet \mid b, \ell = v \\
\text{config types} & \Sigma & ::= & \{\mathbf{w}_1 : \tau_1, \cdots, \mathbf{w}_i : \tau_i\} \\
\text{table types} & \tau & ::= & \bullet \mid \tau, \ell : A \\
\text{world exps} & \mathrm{w} & ::= & \mathbf{w} \mid \omega \\
\text{world vars} & \omega & & \text{world names} \quad \mathbf{w} \\
\text{labels} & \ell & & \text{value vars} \quad x, y
\end{array}
\qquad
\begin{array}{rcll}
\text{values} & v & ::= & \lambda x.M \mid \mathtt{box}\ \omega.M \mid \mathbf{w}.\ell \\
\text{cont stacks} & C & ::= & \star \mid C{::}k \\
\text{conts} & k & ::= & \mathtt{return}\,\mathbf{w} \mid \mathtt{finish} \mid k \lhd f \\
\text{frames} & f & ::= & \circ\, N \mid v\ \circ \mid \mathtt{here}\,\circ \mid \mathtt{unbox}\,\circ \\
& & \mid & \mathtt{letd}\,\omega.x = \circ\ \mathtt{in}\, N \\
\text{exps} & M,N & ::= & v \mid MN \mid x \mid \ell \mid \mathtt{fetch}[\mathrm{w}]M \\
& & \mid & \mathtt{here}\, M \mid \mathtt{get}\ \langle \mathrm{w}\rangle M \\
& & \mid & \mathtt{unbox}\, M \mid \mathtt{letd}\,\omega.x = M\ \mathtt{in}\, N
\end{array}
$$

Figure 3. Syntax of Lambda 5 type system

$$
\frac{}{\Sigma \vdash \mathbb{W}; \mathtt{finish} : A@\mathbf{w}}
\qquad
\frac{\Sigma \vdash \mathbb{W}; k : \Diamond A@\mathbf{w}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{here}\,\circ : A@\mathbf{w}}
\qquad
\frac{\Sigma \vdash \mathbb{W}; k : A'@\mathbf{w} \quad \Sigma; \cdot; \cdot \vdash N : A@\mathbf{w}}{\Sigma \vdash \mathbb{W}; k \lhd \circ\, N : A \supset A'@\mathbf{w}}
$$

$$
\frac{\Sigma \vdash \ell : A@\mathbf{w} \quad \Sigma; \Omega \vdash \mathrm{w}}{\Sigma; \Omega; \Gamma \vdash \mathbf{w}.\ell : \Diamond A@\mathrm{w}}\ \text{dia}
\qquad
\frac{\Sigma \vdash \mathbb{W}; k : A@\mathbf{w}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{unbox}\,\circ : \Box A@\mathbf{w}}
\qquad
\frac{\Sigma \vdash \mathbb{W}; k : B@\mathbf{w} \quad \Sigma; \omega; x : A@\omega \vdash N : B@\mathbf{w}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{letd}\,\omega.x = \circ\ \mathtt{in}\, N : \Diamond A@\mathbf{w}}
$$

$$
\frac{\Sigma \vdash \ell : A@\mathbf{w}}{\Sigma; \Omega; \Gamma \vdash \ell : A@\mathbf{w}}\ \text{lab}
\qquad
\frac{\Sigma \vdash \mathbb{W}; k : B@\mathbf{w} \quad \Sigma; \cdot; \cdot \vdash v : A \supset B@\mathbf{w}}{\Sigma \vdash \mathbb{W}; k \lhd v\ \circ : A@\mathbf{w}}
\qquad
\frac{\Sigma \vdash \{\mathbf{w}' : \langle C; b\rangle; \mathrm{w}_s\}; k : A@\mathbf{w}'}{\Sigma \vdash \{\mathbf{w}' : \langle C{::}k; b\rangle; \mathrm{w}_s\}; \mathtt{return}\,\mathbf{w}' : A@\mathbf{w}}
$$

Figure 4. Extended expression and continuation typing rules

is the rule for `return` **w**. This rule ensures that the continuation stack at **w** is non-empty, and that its outermost continuation expects the same type as the `return`. Via this rule the continuation typing condition *unwinds* the entire network-wide continuation. Also worth noting is that the `finish` continuation is well-formed regardless of any junk that may remain in the continuation stacks in the rest of the network. (This is an arbitrary choice and does not affect type safety.)

$$
\frac{\begin{array}{l}
\Sigma = \{\mathbf{w}_1 : \tau_1, \cdots, \mathbf{w}_i : \tau_i\} \qquad 1 \le j \le i \\
\mathbb{W} = \{\mathbf{w}_1 : \langle C_1, b_1\rangle, \cdots, \mathbf{w}_i : \langle C_i, b_i\rangle\} \\
\Sigma \vdash b_1@\mathbf{w}_1 \quad \cdots \quad \Sigma \vdash b_i@\mathbf{w}_i \\
\Sigma; \cdot; \cdot \vdash M : A@\mathbf{w}_j \qquad \Sigma \vdash \mathbb{W}; k : A@\mathbf{w}_j
\end{array}}{\Sigma \vdash \mathbb{W}; \mathbf{w}_j : [k, M]}
$$

Figure 5. Network typing

Finally, we have the network typing judgment (figure 5). The network $\mathbb{W}; \mathbf{w}_j : [k, M]$ is well formed under some config type Σ if several conditions hold. Both $\mathbb{W}$ and Σ must have the same domain, and $\mathbf{w}_j$ must be in that domain. Each of the tables in $\mathbb{W}$ must be well-formed, and there must exist a mediating type A such that the current expression M has that type and the current continuation k expects it.

With the typing rules in hand, we can give a dynamic semantics to network states that explains the evaluation of distributed programs. Our dynamic semantics takes the form of a stepping relation $\mapsto$ that relates pairs of network states. Its definition is given in figure 6.

Much of the dynamic semantics is standard for a continuation-based abstract machine. The reduction rule for `unbox` (10) instantiates the mobile code with the current world. When we encounter a label (11), we look it up in the current world's table and proceed with that value. To publish a value (9), we generate a new label and add the mapping to our table. The resulting address is our current world paired with the label.

The reduction for `letd` (13) substitutes both that world constant and the disembodied label into the body of the `letd`. Note that our substitution must work on expressions, namely labels. We can't evaluate ℓ yet because we are not necessarily in the correct world.

Finally, the RPC rules are interesting. Evaluating a $\mathtt{fetch}[\mathbf{w}']M$ at **w** (7) means saving the current continuation at **w**, and beginning a new continuation to evaluate M at $\mathbf{w}'$ with `return` **w** at its bottom. The rule for `get` (8) is essentially the same. Reducing `return` **w** (6) simply moves the value to **w**, resuming with its outermost continuation. Only boxes and addresses can be moved.

A programming language is only sensible if it is type safe, that is, if a well-typed program has a defined meaning in terms of evaluation on the abstract machine. In the next section we give the type safety theorem. We then give

$$
\begin{array}{ll}
(1) & \mathbb{W};\mathbf{w}:[k, M N] \;\mapsto \mathbb{W};\mathbf{w}:[k \lhd \circ\ N; M] \\
(2) & \mathbb{W};\mathbf{w}:[k \lhd \circ\ N; v] \;\mapsto \mathbb{W};\mathbf{w}:[k \lhd v\ \circ, N] \\
(3) & \mathbb{W};\mathbf{w}:[k \lhd (\lambda x.M)\circ, v] \;\mapsto \mathbb{W};\mathbf{w}:[k, [v/x]M] \\
(4) & \mathbb{W};\mathbf{w}:[k, \texttt{here}\, M] \;\mapsto \mathbb{W};\mathbf{w}:[k \lhd \texttt{here}\circ, M] \\
(5) & \mathbb{W};\mathbf{w}:[k, \texttt{unbox}\, M] \;\mapsto \mathbb{W};\mathbf{w}:[k \lhd \texttt{unbox}\circ, M] \\
(6) & \{\mathbf{w}:\langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}':[\texttt{return}\,\mathbf{w}, v] \quad \mapsto \\
 & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, v] \quad (v = \texttt{box}\ \omega.M \text{ or } \mathbf{w}''.\ell) \\
(7) & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, \texttt{fetch}[\mathbf{w}']M] \quad \mapsto \\
 & \{\mathbf{w}:\langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}':[\texttt{return}\,\mathbf{w}, M] \\
(8) & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, \texttt{get}\,\langle\mathbf{w}'\rangle M] \quad \mapsto \\
 & \{\mathbf{w}:\langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}':[\texttt{return}\,\mathbf{w}, M] \\
(9) & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k \lhd \texttt{here}\circ, v] \quad \mapsto \\
 & \{\mathbf{w}:\langle C; b, \ell = v\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, \mathbf{w}.\ell] \quad (\ell \text{ fresh}) \\
(10) & \mathbb{W};\mathbf{w}:[k \lhd \texttt{unbox}\circ, \texttt{box}\ \omega.M] \quad \mapsto \\
 & \mathbb{W};\mathbf{w}:[k, [\mathbf{w}/\omega]M] \\
(11) & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, \ell] \quad \mapsto \\
 & \{\mathbf{w}:\langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}:[k, v] \quad (b(\ell) = v) \\
(12) & \mathbb{W};\mathbf{w}:[k, \texttt{letd}\,\omega.x = M\ \texttt{in}\,N] \quad \mapsto \\
 & \mathbb{W};\mathbf{w}:[k \lhd \texttt{letd}\,\omega.x = \circ\ \texttt{in}\,N, M] \\
(13) & \mathbb{W};\mathbf{w}:[k \lhd \texttt{letd}\,\omega.x = \circ\ \texttt{in}\,N, \mathbf{w}'.\ell] \quad \mapsto \\
 & \mathbb{W};\mathbf{w}:[k, [\ell/x][\mathbf{w}'/\omega]N]
\end{array}
$$

Figure 6. Dynamic Semantics

a comparison to a hypothetical system where the rules act non-locally.

4.3 Type Safety

Type safety is the conjunction of two properties, progress (theorem 5) and type preservation (theorem 6). Progress states that any well-formed network state is either *terminal* (meaning it has successfully finished computation) or can make a step to a new network state. Preservation states that any step we make from a well-formed network results in a state that is also well-formed. A network is terminal if it is of the form $\mathbb{W};\mathbf{w}:[\texttt{finish}, v]$. We say that store types are related as $\Sigma \supseteq \Sigma'$ if they have the same world constants in their domains, and for each world the table types $\tau = \Sigma(\mathbf{w}_i)$ and $\tau' = \Sigma'(\mathbf{w}_i)$ agree on the domain of τ.

Theorem 5 (Progress)
If $\quad \mathcal{D} :: \Sigma \vdash \mathbb{N}$
then $\quad$ *either* $\mathbb{N}$ *is terminal* $\quad$ *or* $\quad \exists \mathbb{N}'.\mathbb{N} \mapsto \mathbb{N}'$.

Theorem 6 (Preservation)
If $\quad \mathcal{D} :: \Sigma \vdash \mathbb{N}$ *and* $\mathcal{E} :: \mathbb{N} \mapsto \mathbb{N}'$
then $\quad \exists \Sigma', \mathcal{F}.\Sigma' \supseteq \Sigma \quad$ *and* $\quad \mathcal{F} :: \Sigma' \vdash \mathbb{N}'$.

Proof of progress is by induction on the derivation $\mathcal{D}$. Proof of preservation is by induction on the derivation $\mathcal{E}$ with inversions on $\mathcal{D}$. These proofs are fairly standard and appear in the companion report.

Therefore, a well typed program can make a step (or is done), and steps to another well-typed program. By iterating these two theorems it is easy to see that a well-typed program can never become stuck. [6]

4.4 Comparison

To justify our decomposition, we compare the proof terms from section 4.1 to a hypothetical system "H5" where the rules act non-locally (closely modeled after Simpson's system $\mathbf{N}_{\Box\Diamond}$ [17]). It shares features with calculi discussed in section 6.

H5 has no `get` or `fetch`; instead it replaces `here`, `unbox`, and `letd` with three new terms:

- $\texttt{there}\,\langle\omega\rangle M$, which computes M of type A at ω and then returns its address of type $\Diamond A$;
- $\texttt{unboxfrom}[\omega]M$, which computes M (of type $\Box A$) at ω, and then returns its value of type A;
- $\texttt{letdfrom}\,\langle\omega\rangle\ \omega'.y = M\ \texttt{in}\,N$, which is like `letd` except that it computes M (of type $\Diamond A$) at ω instead of locally.

In H5, the proof term of $\Diamond\Box A \supset \Box A@\omega$ would be:

$$
\text{(H5)} \qquad \begin{array}{l} \lambda x.\,\texttt{letdfrom}\,\langle\omega\rangle\ \omega'.y = x \\ \quad \texttt{in box}\ \omega''.\,\texttt{unboxfrom}[\omega]\ y \end{array}
$$

Note that this term is not moving the code at all! Instead, it creates a new box that, when opened, will unbox the code from the original world into the target world. This hardly fits our model of mobile code. Moreover, the $\Diamond$ elimination `letdfrom` allows its source to be an arbitrary world, so we may end up calling ourselves remotely. An implementation could optimize local RPC, but it is better to enable purely local reasoning in the semantics itself.

The H5 proof term of $\Diamond\Diamond A \supset \Diamond A@\omega$ is:

$$
\text{(H5)} \qquad \begin{array}{l} \lambda r.\,\texttt{letdfrom}\,\langle\omega\rangle\ \omega'.x = r \\ \quad \texttt{in letdfrom}\,\langle\omega'\rangle\ \omega''.y = x \\ \qquad \texttt{in there}\,\langle\omega''\rangle y \end{array}
$$

In addition to the self-RPC seen in the last term, the H5 program is forced to deconstruct both diamonds and reintroduce a direct address. This has the effect of publishing A in the table at ω'', where it already must have been published!

[6] However, as stated our type safety theorem does not guarantee that the type of the final value sent to `finish` does not change through the course of execution. To prove this we can index the network well-formedness judgment with the "final answer" type and modify the continuation typing rule for `finish` without any change in the preservation proof, observing that none of the transitions modify this type.

5 Future Work

With the minimal set of connectives presented here, our system has the same consequence relation as Simpson's IS5. This is because the accessibility relation in S5 is that of equivalence classes. Although there may be more than one equivalence class of worlds, disjoint classes cannot affect each other. Now, Lambda 5 only supports reasoning about a single class; the list of worlds in Ω. Each IS5 theorem is proved at some world, and so we can focus our attention on that world's class and repeat the proof in Lambda 5, discarding any assumptions from other classes.

The addition of some other standard connectives like $\wedge$ and $\top$ poses no problem. When introducing disjunctive connectives like $\perp$ and $\vee$, however, we must be careful. Compare the elimination rule for $\Box$ with the elimination for $\perp$ in Simpson's IS5:

$$\frac{\Box A@\omega \quad \omega\ R\ \omega'}{A@\omega'}\ \Box E \qquad\qquad \frac{\perp@\omega}{C@\omega'}\ \perp E$$

Here, $\omega\ R\ \omega'$ if ω' is accessible from ω. In order to unbox from one world into another they must be in the same equivalence class. However, if $\perp$ is true at some world then any proposition is true at *any other world*, irrespective of their mutual (in)accessibility. Now our argument above does not hold, because disjoint equivalence classes may affect each other. In the presence of $\perp$ or $\vee$ we must make the slightly weaker claim that IS5 and Lambda 5 have the same consequence relation under assumptions about a single class only. This includes all relations of the form $\omega; \cdot \vdash A@\omega$ because all worlds introduced in the proof of $A@\omega$ will be interaccessible with ω.

Because $\perp$ and $\vee$ reason non-locally, we require special considerations in the operational semantics. Falsehood is simple: since there is no value of type $\perp E$ we can initiate a remote procedure call which is known never to return. For $\vee$, the value analyzed is not generally portable to our world. We conjecture that a remote procedure call mechanism can distinguish cases remotely and send back only a label and a bit indicating whether the left of right case applies.

By design, our operational semantics is sequential. However, many distributed computing tasks rely essentially on concurrency. In order to develop Lambda 5 into a realistic programming language, we intend to add support for concurrency. We believe this will be an orthogonal extension. Other programming constructs such as recursion, polymorphism, and other type constructs for functional programming should also be easily added. On the implementation side, we need to consider details such as distributed garbage collection, failure recovery, as well as marshalling and certification of mobile code.

6 Related Work

Others have also used modal logic for distributed computing. For example, Borghuis and Feijs's Modal Type System for Networks [1] provides a logic and operational semantics[7] for network tasks with stationary services and mobile data. They use $\Box$, annotated with a location, to represent services. For example, $\Box^o(A \supset B)$ means a function from A to B at the location o. With no way of internalizing mobility as a proposition, the calculus limits mobile data to base types. Services are similarly restricted to depth-one arrow types. By using $\Box$ for mobile code and $\Diamond$ for stationary resources, we believe our resulting calculus is both simpler and more general.

Cardelli and Gordon [4] provide an early example of using modal logic for reasoning about programs spatially, later refined by Caires and Cardelli [2, 3]. They do not take a propositions-as-types view of their logic; instead, they start from a process calculus, mobile ambients, and develop a classical logic for reasoning about their behaviors. Therefore, their modal logic is very different from intuitionistic S5 and includes connectives for stating temporal properties, security properties, and properties of parallel compositions. In contrast, Lambda 5 may be seen as a pure study of mobility and locality in a fully interconnected network.

Hennessy et al. [5] develop a distributed version of the π-calculus and impose a complex static type system in order to constrain and describe behavior. Similarly, Schmitt and Stefani [15] develop a distributed, higher-order version of the Join Calculus with a complex behavioral type system. In comparison, our system is much simpler, eliminating the complexities of concurrency, access control, and related considerations. By basing our system on the Curry-Howard correspondence, we have a purely logical analysis and, furthermore, we expect straightforward integration into a full-scale functional language for realistic programs.

Moody [9] gives a system based on the constructive modal logic S4 due to Pfenning and Davies [12]. This language is based on judgments $A\,\mathsf{true}$ (here), $A\,\mathsf{poss}$ (somewhere), and $A\,\mathsf{valid}$ (everywhere) rather than truth at particular worlds. The operational semantics of his system takes the form of a process calculus with nondeterminism, concurrency and synchronization; a significantly different approach from our sequential abstract machine. From the standpoint of a multiple world semantics, the accessibility relation of S4 satisfies only reflexivity and transitivity, not symmetry. From the computational point of view, accessibility describes process interdependence rather than connections between actual network locations. Programs are therefore somewhat higher-level and express *potential mobility* instead of explicit code motion as in the *fetch* and *get* constructs. In particular, due to the lack of symmetry it is

[7] By way of compilation into shell scripts!

not possible to go back to a source world after a potentially remote procedure call except by returning a value.

Jia and Walker [6] give a judgmental account of an S5-like system based on hybrid logics, but compare it only informally to known logics. Hybrid logics internalize worlds inside propositions by including a *proposition* that a value of type A resides at world ω, "$A\,\mathrm{at}\,\omega$." This leads to a technically different logic and language though they have similar goals. Their rules for $\Box$ and $\Diamond$ are similar to the non-local H5 system that we compare Lambda 5 to in section 4.4. Like Moody, they give their network semantics as a process calculus with passive synchronization across processes as a primitive notion. In comparison, we are able to achieve active returns of values by restricting our non-local computation to two terms, and associating remote labels with entries in a table rather than with processes. We feel that this is a more realistic and efficient semantics.

7 Conclusion

We have presented a logic and foundational programming language Lambda 5 for distributed computation based on a Curry-Howard isomorphism for the intuitionistic modal logic S5, viewed from a multiple-world perspective. Computationally, values of type $\Box A$ are mobile code and values of type $\Diamond A$ are addresses of remote values, providing a type-theoretic analysis of mobility and locality in an interconnected network. We have shown that Lambda 5 remains faithful to the logic, via translations from natural deduction to and from a sequent calculus in which cut is admissible. Moreover, by localizing introduction and elimination rules for mobile and remote code ($\Box E$, $\Diamond I$, and $\Diamond E$) and adding explicit rules for code motion, we achieve an efficient and natural computational interpretation.

References

[1] Tijn Borghuis and Loe M. G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000.

[2] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). In *Theoretical Aspects of Computer Software (TACS)*, pages 1–37. Springer-Verlag LNCS 2215, October 2001.

[3] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part II). In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR)*, pages 209–225, Brno, Czech Republic, August 2002. Springer-Verlag LNCS 2421.

[4] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM Press, 2000.

[5] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDPi: A language for controlling mobile code. Report 02/2003, Department of Computer Science, University of Sussex, October 2003.

[6] Limin Jia and David Walker. Modal proofs as distributed programs. *13th European Symposium on Programming*, pages 219–223, March 2004.

[7] S. Kanger. *Provability in Logic*. Almquist and Wiksell, Stockholm, 1957.

[8] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[9] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, Oct 2003.

[10] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. Technical Report CMU-CS-04-105, Carnegie Mellon University, Mar 2004.

[11] Frank Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.

[12] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[13] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag. LNAI 1632.

[14] A. N. Prior. *Time and Modality*. Oxford University Press, 1957.

[15] Alan Schmitt and Jean-Bernard Stefani. The M-calculus: A higher-order distributed process calculus. In *Conference Record of the 30th Symposium on Principles of programmming Languages*, pages 50–61, New Orleans, Louisiana, January 2003. ACM Press.

[16] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

[17] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.