A Symmetric Modal Lambda Calculus for Distributed Computing [1]

Tom Murphy VII Karl Crary Robert Harper
Frank Pfenning

March 1, 2004
CMU-CS-04-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a foundational language for 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] The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination."

Keywords: distributed computing, lambda calculus, modal logic, curry-howard isomorphism, lambda 5, type theory, programming languages

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 distributed computing. Scientific computing tasks often require the physical distribution of computational resources and sensing instruments. 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. We argue that intuitionistic modal logic forms an excellent basis for distributed computing because of its ability to represent spatial reasoning.

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 [15]. 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.

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 [15], 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 useful computational content to a proof that two worlds are related. We therefore dispense with judgments of the accessibility relation 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 4.5 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.

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 strong 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.

In this extended technical report we provide interesting cases of the proofs. Most proofs have been mechanized in the Twelf system [12] and verified using its metatheorem checker [14]. They appear in appendix B and electronically 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

$$\frac{\Omega; \Gamma, A@\omega \vdash A'@\omega}{\Omega; \Gamma \vdash A \supset A'@\omega} \supset I \qquad \frac{\begin{array}{c}\Omega; \Gamma \vdash A'@\omega \\ \Omega; \Gamma \vdash A' \supset A@\omega\end{array}}{\Omega; \Gamma \vdash A@\omega} \supset E \qquad \frac{\omega \in \Omega}{\Omega; \Gamma, A@\omega, \Gamma' \vdash A@\omega} \text{ hyp}$$

$$\frac{\omega' \text{ fresh} \quad \Omega, \omega'; \Gamma \vdash A@\omega' \quad \omega \in \Omega}{\Omega; \Gamma \vdash \Box A@\omega} \Box I \qquad \frac{\Omega; \Gamma \vdash \Box A@\omega}{\Omega; \Gamma \vdash A@\omega} \Box E \qquad \frac{\omega \in \Omega \quad \Omega; \Gamma \vdash \Diamond A@\omega'}{\Omega; \Gamma \vdash \Diamond A@\omega} \text{ get}$$

$$\frac{\begin{array}{c}\omega' \text{ fresh} \quad \Omega; \Gamma \vdash \Diamond A@\omega \\ \Omega, \omega'; \Gamma, A@\omega' \vdash B@\omega\end{array}}{\Omega; \Gamma \vdash B@\omega} \Diamond E \qquad \frac{\Omega; \Gamma \vdash A@\omega}{\Omega; \Gamma \vdash \Diamond A@\omega} \Diamond I \qquad \frac{\omega \in \Omega \quad \Omega; \Gamma \vdash \Box A@\omega'}{\Omega; \Gamma \vdash \Box A@\omega} \text{ fetch}$$

Figure 1: Lambda 5 natural deduction

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 $\omega_i \ \mathsf{exists}$ 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.[1]

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

[1] 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.

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.

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 $\mathcal{D}::\Omega; \Gamma \vdash A@\omega$
and $\mathcal{E}::\Omega; \Gamma, A@\omega \vdash B@\omega'$
then $\mathcal{F}::\Omega; \Gamma \vdash B@\omega'$.

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 $\mathcal{E} :: \Omega, \omega; \Gamma \vdash A@\omega''$
then $\mathcal{F} :: [\omega'/\omega](\Omega; \Gamma \vdash A@\omega'')$

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.

For each connective, we require the properties of local soundness and completeness. Local soundness ensures us 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. This is also called a *local reduction*.

The reduction for $\Box$ is as follows:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega, \omega, \omega'; \Gamma \vdash A@\omega'\end{array}}{\Omega, \omega; \Gamma \vdash \Box A@\omega}\,\Box I}{\Omega, \omega; \Gamma \vdash A@\omega}\,\Box E \quad \Rightarrow_R \quad \begin{array}{c}[\omega/\omega']\mathcal{D}\\ \vdots\\ \Omega, \omega; \Gamma \vdash A@\omega\end{array}$$

If we derive A at a hypothetical fresh world ω' (calling this derivation $\mathcal{D}$), and then use $\Box A$ to conclude $A@\omega$, we can reduce this to a direct use of $\mathcal{D}$ by our world substitution principle, abbreviated here as $[\omega/\omega']\mathcal{D}$.

The reduction for $\Diamond$ is similar, employing both world and regular substitution:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega; \Gamma \vdash A@\omega\end{array}}{\Omega; \Gamma \vdash \Diamond A@\omega}\,\Diamond I \qquad \begin{array}{c}\mathcal{E}\\ \vdots\\ \Omega, \omega'; \Gamma, x{:}A@\omega' \vdash B@\omega\end{array}}{\Omega; \Gamma \vdash B@\omega}\,\Diamond E$$

$$\Rightarrow_R$$

$$\begin{array}{c}[\mathcal{D}/x][\omega/\omega']\mathcal{E}\\ \vdots\\ \Omega; \Gamma \vdash B@\omega\end{array}$$

Here we write $[\mathcal{D}/x]$ for an application of the substitution principle to $[\omega/\omega']\mathcal{E}$, which is itself the result of world substitution.

Note that neither local reduction accounts for the motion rules *get* and *fetch*. The global soundness property in section 3 shows that this is not problematic. We omit the standard reduction for $\supset$. Note that under the Curry-Howard isomorphism the action of the local reductions on proof terms forms the core of the operational semantics.

The counterpart to local soundness is local completeness. This ensures that our elimination rules are not too weak—if we have a derivation of a formula using the connective, we can apply our elimination rules in such a way as to reintroduce the formula. The process is known as a *local expansion*. The expansion for $\Box$ is as follows:

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega, \omega; \Gamma \vdash \Box A@\omega\end{array}$$

$$\Rightarrow_E$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega, \omega, \omega'; \Gamma \vdash \Box A@\omega\end{array}}{\Omega, \omega, \omega'; \Gamma \vdash \Box A@\omega'}\,\text{fetch}}{\Omega, \omega, \omega'; \Gamma \vdash A@\omega'}\,\Box E}{\Omega, \omega; \Gamma \vdash \Box A@\omega}\,\Box I$$

If we have a derivation of $\Box A$ ($\mathcal{D}$) we can reintroduce $\Box A$ by fetching it into the hypothetical world ω' about which nothing else is known, which is enough to apply $\Box I$. The expansion for $\Diamond$ is similar:

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega, \omega; \Gamma \vdash \Diamond A@\omega\end{array}$$

$$\Rightarrow_E$$

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega, \omega; \Gamma \vdash \Diamond A@\omega\end{array} \qquad \dfrac{\dfrac{\dfrac{}{\Omega, \omega, \omega'; \Gamma, A@\omega' \vdash A@\omega'}\,\text{hyp}}{\Omega, \omega, \omega'; \Gamma, A@\omega' \vdash \Diamond A@\omega'}\,\Diamond I}{\Omega, \omega, \omega'; \Gamma, A@\omega' \vdash \Diamond A@\omega}\,\text{get}}{\Omega, \omega; \Gamma \vdash \Diamond A@\omega}\,\Diamond E$$

If we have a derivation of $\Diamond A$ ($\mathcal{D}$) we can reintroduce $\Diamond A$ by learning A at some hypothetical ω' via $\Diamond E$, then reintroducing $\Diamond A$ there and moving it back to ω.

Again, we omit the expansion for $\supset$, which is standard. As implied by their names, local soundness and completeness give us only a local guarantee that our logic makes sense. In fact, local soundness is weaker than usual because of our motion rules. Though we see that $\Box I$ followed by $\Box E$ is justified, what about an intervening sequence of *fetch* rules? The global check 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. The following section proves this correspondence. The operational interpretation (section 4) does not depend on it.

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

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

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

Figure 2: Sequent calculus SS5

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 $\Diamond R$ 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 lexicographic induction on (in order) the cut formula A, the derivation $\mathcal{D}$, and the derivation $\mathcal{E}$, following Pfenning [10]. This proof is new[2] and appears in machine checkable form in appendix B. We present a few characteristic cases here.

We must check each possible combination of rules used to conclude $\mathcal{D}$ and rules used to conclude $\mathcal{E}$. (In practice we do not need to look at all n^2 cases since many can be dealt with schematically.) Each pair takes on one of the following forms, for which we provide a representative case.

[2] Simpson [15] used an indirect proof via natural deduction

Initial Cuts If $\mathcal{D}$ or $\mathcal{E}$ is an initial sequent, we proceed directly.

Example: Suppose $\mathcal{D}$ is any derivation and $\mathcal{E} = \frac{}{\Omega, \omega; \Gamma, A@\omega \longrightarrow A@\omega} \text{init}$. Then $\mathcal{F} = \mathcal{D}$.

Left-Commutative Cuts If $\mathcal{D}$ ends with a use of a left rule, we appeal to the induction hypothesis on the same cut formula but a smaller left derivation. We then re-apply the left rule to the cut-free proof.

Example:

$$\mathcal{D} = \frac{\begin{matrix}\mathcal{D}' \\ \vdots \\ \Omega', \omega^3; \Gamma', \Box C@\omega'', C@\omega^3 \longrightarrow A@\omega\end{matrix}}{\Omega', \omega^3; \Gamma', \Box C@\omega'' \longrightarrow A@\omega} \Box L$$

$\mathcal{E}$ arbitrary

Then

$$\mathcal{F} = \frac{\begin{matrix}IH(\mathcal{D}', \text{weaken-}\Gamma(\mathcal{E})) \\ \vdots \\ \Omega, \omega^3; \Gamma', \Box C@\omega'', C@\omega^3 \longrightarrow B@\omega'\end{matrix}}{\Omega, \omega^3; \Gamma', \Box C@\omega'' \longrightarrow B@\omega'} \Box L$$

Right-Commutative Cuts Similarly, if $\mathcal{E}$ ends with a use of a right rule, or a left rule acting on something other than the cut formula A, then we have a right commutative case. We appeal to the induction hypothesis on the same A but smaller right derivation, and then re-apply the rule to the cut-free proof.

Example:

$\mathcal{D}$ arbitrary

$$\mathcal{E} = \frac{\begin{matrix}\mathcal{E}' \\ \vdots \\ \Omega', \omega'; \Gamma, A@\omega \longrightarrow C@\omega''\end{matrix}}{\Omega', \omega'; \Gamma, A@\omega \longrightarrow \Diamond C@\omega'} \Diamond R$$

Then

$$\mathcal{F} = \cfrac{\begin{array}{c} IH(\mathcal{D},\mathcal{E}') \\ \vdots \\ \Omega', \omega'; \Gamma \longrightarrow C@\omega' \end{array}}{\Omega', \omega'; \Gamma \longrightarrow \Diamond C@\omega'} \Diamond R$$

Principal Cuts If the derivation $\mathcal{D}$ just concluded the cut formula A with a right rule and the derivation $\mathcal{E}$ just made use of A with the corresponding left rule, then we have a principal cut. Our strategies differ depending on the connective in question, but we always appeal to the inductive hypothesis on A (but smaller derivations $\mathcal{D}$ and $\mathcal{E}$) and then on subformulas of A and larger derivations. Because these are the most interesting cases we show both the $\Box$ and $\Diamond$ principal cuts.

Example 1:

$$\mathcal{D} = \cfrac{\begin{array}{c} \mathcal{D}' \\ \vdots \\ \Omega, \omega, \omega'', \omega^3; \Gamma \longrightarrow A@\omega^3 \end{array}}{\Omega, \omega, \omega''; \Gamma \longrightarrow \Box A@\omega} \Box R$$

$$\mathcal{E} = \cfrac{\begin{array}{c} \mathcal{E}' \\ \vdots \\ \Omega, \omega, \omega''; \Gamma, \Box A@\omega, A@\omega'' \longrightarrow B@\omega' \end{array}}{\Omega, \omega, \omega''; \Gamma, \Box A@\omega \longrightarrow B@\omega'} \Box L$$

1.	$\Omega, \omega, \omega''; \Gamma, A@\omega'' \longrightarrow \Box A@\omega$	(weakening $\mathcal{D}$)
2.	$\Omega, \omega, \omega''; \Gamma, A@\omega'' \longrightarrow B@\omega'$	(IH(1,$\mathcal{E}'$))
3.	$\Omega, \omega, \omega''; \Gamma \longrightarrow A@\omega''$	($[\omega''/\omega^3]\mathcal{D}'$)
4.	$\Omega, \omega, \omega''; \Gamma \longrightarrow B@\omega'$	(IH(3, 2, $A@\omega$))

The final appeal to the induction hypothesis is justified by the smaller cut formula.

Example 2:

$$\mathcal{D} = \cfrac{\begin{array}{c} \mathcal{D}' \\ \vdots \\ \Omega, \omega; \Gamma \longrightarrow A@\omega^3 \end{array}}{\Omega, \omega; \Gamma \longrightarrow \Diamond A@\omega} \Diamond R$$

$$\mathcal{E} = \cfrac{\begin{array}{c} \mathcal{E}' \\ \vdots \\ \Omega, \omega, \omega''; \Gamma, \Diamond A@\omega, A@\omega'' \longrightarrow B@\omega' \end{array}}{\Omega, \omega; \Gamma, \Diamond A@\omega \longrightarrow B@\omega'} \Diamond L$$

1.	$\Omega, \omega, \omega''; \Gamma, A@\omega'' \longrightarrow \Diamond A@\omega$	(weakening $\mathcal{D}$)
2.	$\Omega, \omega, \omega''; \Gamma, A@\omega'' \longrightarrow B@\omega'$	(IH(1, $\mathcal{E}'$))
3.	$\Omega, \omega; \Gamma, A@\omega^3 \longrightarrow B@\omega'$	($[\omega^3/\omega'']2$)
4.	$\Omega, \omega; \Gamma \longrightarrow B@\omega'$	IH($\mathcal{D}'$, 3, $A@\omega^3$)

World substitution in step 3 requires that $\omega^3 \in \Omega$. This is justified by our restriction of judgments to those where all mentioned world variables appear in Ω. Our appeal to the induction hypothesis in step 4 is again justified by the reduction in size of the cut formula. $\Box$

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 verification, has a verification where the *init* rule is applied only to atomic propositions (theorem 4). Second, we have to show that every proposition that has a proof (in natural deduction) has a verification (in the sequent calculus). This (theorem 5) is the global analogue of the local soundness property.

Theorem 4 (Global Completeness) *For any A,* $\Omega, \omega; \Gamma, A@\omega \longrightarrow^* A@\omega$.

In $\longrightarrow^*$, the *init* rule is replaced with

$$\cfrac{}{\Omega, \omega; \Gamma, p@\omega \longrightarrow^* p@\omega} \text{initp}$$

where p is an atomic proposition. Proof is by induction on the structure of A. For example:

Case $A = \Box B$.

$$\cfrac{\cfrac{\begin{array}{c} IH(B) \\ \vdots \\ \Omega, \omega, \omega'; \Gamma, \Box B@\omega, B@\omega' \longrightarrow^* B@\omega' \end{array}}{\Omega, \omega, \omega'; \Gamma, \Box B@\omega \longrightarrow^* B@\omega'} \Box L}{\Omega, \omega; \Gamma, \Box B@\omega \longrightarrow^* \Box B@\omega} \Box R$$

Global completeness is the global analogue of the local completeness property, because it ensures that the left rules are strong enough to decompose and reconstitute any proposition.

Theorem 5 (Equivalence of Lambda 5 and SS5)

$$\Omega; \Gamma \vdash A@\omega \quad \textit{iff} \quad \Omega; \Gamma \longrightarrow A@\omega.$$

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. The interesting cases come from the mismatch between the rules for $\Box$ and $\Diamond$ in the two systems. We provide a few cases here and again note that the proof is given in machine-checkable form in appendix B.

If $\mathcal{D} :: \Omega; \Gamma \vdash A@\omega$ **then** $\mathcal{F} :: \Omega; \Gamma \longrightarrow A@\omega$**.**
Case 1:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega; \Gamma \vdash \Box A@\omega'\end{array}}{\Omega, \omega; \Gamma \vdash \Box A@\omega}\ \text{fetch}$$

Then let

$$\mathcal{E} = \dfrac{\dfrac{\overline{\Omega, \omega, \omega''; \Gamma, \Box A@\omega', A@\omega'' \longrightarrow A@\omega''}\ \text{init}}{\Omega, \omega, \omega''; \Gamma, \Box A@\omega' \longrightarrow A@\omega''}\ \Box L}{\Omega, \omega; \Gamma, \Box A@\omega' \longrightarrow \Box A@\omega}\ \Box R$$

1. $\Omega, \omega; \Gamma \longrightarrow \Box A@\omega'$ (IH $\mathcal{D}'$)
2. $\Omega, \omega; \Gamma \longrightarrow \Box A@\omega$ (cut(1, $\mathcal{E}$))

Case 2:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega; \Gamma \vdash \Diamond A@\omega'\end{array}}{\Omega, \omega; \Gamma \vdash \Diamond A@\omega}\ \text{get}$$

Then let

$$\mathcal{E} = \dfrac{\dfrac{\overline{\Omega, \omega, \omega''; \Gamma, \Diamond A@\omega', A@\omega'' \longrightarrow A@\omega''}\ \text{init}}{\Omega, \omega, \omega''; \Gamma, \Diamond A@\omega', A@\omega'' \longrightarrow \Diamond A@\omega}\ \Diamond R}{\Omega, \omega; \Gamma, \Diamond A@\omega' \longrightarrow \Diamond A@\omega}\ \Diamond L$$

1. $\Omega, \omega; \Gamma \longrightarrow \Diamond A@\omega'$ (IH $\mathcal{D}'$)
2. $\Omega, \omega; \Gamma \longrightarrow \Diamond A@\omega$ (cut(1, $\mathcal{E}$))

The remainder of the cases are straightforward given the cut theorem. $\Box$

If $\mathcal{D} :: \Omega; \Gamma \longrightarrow A@\omega$ **then** $\Omega; \Gamma \vdash A@\omega$
Case 1:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega'; \Gamma, \Box A@\omega, A@\omega' \longrightarrow C@\omega''\end{array}}{\Omega, \omega'; \Gamma, \Box A@\omega \longrightarrow C@\omega''}\ \Box L$$

Then let

$$\mathcal{E} = \dfrac{\dfrac{\overline{\Omega, \omega'; \Gamma, \Box A@\omega \vdash \Box A@\omega}\ \text{hyp}}{\Omega, \omega'; \Gamma, \Box A@\omega \vdash \Box A@\omega'}\ \text{fetch}}{\Omega, \omega'; \Gamma, \Box A@\omega \vdash A@\omega'}\ \Box E$$

1. $\Omega, \omega'; \Gamma, \Box A@w, A@\omega' \vdash C@\omega''$ (IH $\mathcal{D}'$)
2. $\Omega, \omega'; \Gamma, \Box A@w \vdash C@\omega''$ (subst($\mathcal{E}$, 1))

Case 2:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega; \Gamma \longrightarrow A@\omega'\end{array}}{\Omega, \omega; \Gamma \longrightarrow \Diamond A@\omega}\ \Diamond R$$

Then

$$\mathcal{F} = \dfrac{\dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega; \Gamma \vdash A@\omega'\end{array}}{\Omega, \omega; \Gamma \vdash \Diamond A@\omega'}\ \Diamond I}{\Omega, \omega; \Gamma \vdash \Diamond A@\omega}\ \text{get}$$

Case 3:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Omega, \omega'; \Gamma, \Diamond A@\omega, A@\omega' \longrightarrow C@\omega''\end{array}}{\Omega; \Gamma, \Diamond A@\omega \longrightarrow C@\omega''}\ \Diamond L$$

Then

$$\mathcal{F} = \dfrac{\dfrac{\overline{\Omega; \Gamma, \Diamond A@\omega \vdash \Diamond A@\omega}\ \text{hyp}}{\Omega; \Gamma, \Diamond A@\omega \vdash \Diamond A@\omega''}\ \text{get} \qquad \mathcal{F}'}{\Omega; \Gamma, \Diamond A@\omega \vdash C@\omega''}\ \Diamond E$$

Where

$$\mathcal{F}' = \begin{array}{c} IH(\mathcal{D}') \\ \vdots \\ \Omega, \omega'; \Gamma, \Diamond A@\omega, A@\omega' \vdash C@\omega''\end{array}$$

The remainder of the cases are straightforward given the substitution principle. $\Box$

In this proof the dualities between $\Box$ and $\Diamond$ are somewhat obscured. First note that the case for $\Box R$ (not shown) is immediate by induction, whereas the case for $\Diamond R$ requires an intervening *get*. Because the $\Box R$ rule's premise is at a new hypothetical world we could think of it as being non-local reasoning, and insert a *fetch*, making the duality clear. This simply leads to slightly longer proofs. Secondly, the $\Box L$ rule requires a substitution while the $\Diamond L$ rule does not. This is simply because the *let* form of $\Diamond$ elimination has a built-in substitution. We could have equivalently considered a *let* form for $\Box E$, which would have made these cases match up.

We can exploit the computational content of this meta-theoretic proof to translate an arbitrary natural deduction to the sequent calculus and then back. Analysis of the proofs of theorem 5 shows that the resulting natural deduction will satisfy a strong 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 consistency: 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 [15].

In order to bridge the gap from logic to programming language, we give a proof term assignment to Lambda 5 natural deduction in the next section, which is then given a distributed operational semantics.

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 add proof terms to the natural deduction system from section 2 (figure 3). 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 $\mathbf{w}$ 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 attempt to evaluate under the $\texttt{box}$. Straightforwardly, $\texttt{unbox}$ instantiates the hypothetical world with the actual current world and then evaluates the contents of the $\texttt{box}$. 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 $\texttt{fetch}$ 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 $\texttt{box}$ 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, regardless of the accessibility relation.[3] Here is the proof term, assuming that it lives at ω.

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

[3]However, it is not provable in *constructive* modal logics such as the judgmental S4 due to Pfenning and Davies [11] where necessity is taken to mean provability with *no* assumptions.

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

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

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

Figure 3: Lambda 5 external language

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:

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

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

In section 4.5 we justify our decomposition by comparing some of these proof terms to a hypothetical system where the rules act non-locally.

4.2 Local Reductions and Expansions With Proof Terms

We can repeat the local reductions and expansions from section 2 in the presence of proof terms. The reductions form the basis of our operational semantics, so it is interesting to see the computational principles involved. The reduction for $\Box$ is as follows:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega,\omega,\omega';\Gamma\vdash M:\omega'@\end{array}}{\Omega,\omega;\Gamma\vdash\texttt{box}\ \omega'.M:\Box A@\omega}\Box I}{\Omega,\omega;\Gamma\vdash\texttt{unbox}\,\texttt{box}\ \omega'.M:A@\omega}\Box E$$

$$\Rightarrow_R$$

$$\begin{array}{c}[\omega/\omega']\mathcal{D}\\ \vdots\\ \Omega,\omega;\Gamma\vdash[\omega/\omega']M:A@\omega\end{array}$$

Boxing simply abstracts over a world, which unboxing fills in with the current world.

The reduction for $\Diamond$ is:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega;\Gamma\vdash M:A@\omega\end{array}}{\Omega;\Gamma\vdash\texttt{here}\,M:\Diamond A@\omega}\Diamond I\qquad\begin{array}{c}\mathcal{E}\\ \vdots\\ \Omega,\omega';\Gamma,x{:}A@\omega'\vdash N:B@\omega\end{array}}{\Omega;\Gamma\vdash\texttt{letd}\,\omega'.x=\texttt{here}\,M\,\texttt{in}\,N:B@\omega}\Diamond E$$

$$\Rightarrow_R$$

$$\begin{array}{c}[\mathcal{D}/x][\omega/\omega']\mathcal{E}\\ \vdots\\ \Omega;\Gamma\vdash[M/x][\omega'/\omega]N:B@\omega\end{array}$$

Here we substitute the *actual* world ω at which the expression M exists for the hypothetical world ω'. We also substitute the expression M for the bound variable x.

The expansion for $\Box$ is as follows:

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega,\omega;\Gamma\vdash M:\Box A@\omega\end{array}$$

$$\Rightarrow_E$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega,\omega,\omega';\Gamma\vdash M:\Box A@\omega\end{array}}{\Omega,\omega,\omega';\Gamma\vdash\texttt{fetch}[\omega]M:\Box A@\omega'}\,\text{fetch}}{\Omega,\omega,\omega';\Gamma\vdash\texttt{unbox}\ \texttt{fetch}[\omega]M:A@\omega'}\Box E}{\Omega,\omega;\Gamma\vdash\texttt{box}\ \omega'.\,\texttt{unbox}\ \texttt{fetch}[\omega]M:\Box A@\omega}\Box I$$

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Omega,\omega;\Gamma\vdash M:\Diamond A@\omega\\ \Rightarrow_E\\ \mathcal{D}\\ \vdots\\ \Omega,\omega;\Gamma\vdash M:\Diamond A@\omega \quad \mathcal{E}\end{array}}{\Omega,\omega;\Gamma\vdash \mathtt{letd}\,\omega'.x=M\,\mathtt{in}\,\mathtt{get}\langle\omega'\rangle\,\mathtt{here}\,x:\Diamond A@\omega}\,\Diamond E$$

where $\mathcal{E} =$

$$\dfrac{\dfrac{\dfrac{}{\Omega,\omega,\omega';\Gamma,x{:}A@\omega'\vdash x:A@\omega'}\,\text{hyp}}{\Omega,\omega,\omega';\Gamma,x{:}A@\omega'\vdash \mathtt{here}\,x:\Diamond A@\omega'}\,\Diamond I}{\Omega,\omega,\omega';\Gamma,x{:}A@\omega'\vdash \mathtt{get}\langle\omega'\rangle\,\mathtt{here}\,x:\Diamond A@\omega}\,\text{get}$$

In each of these expansions, we simply wrap the proof term with the primitives that eliminate and then reintroduce it.

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

4.3 Type System

The syntax of our type system and operational semantics is given in figure 4. As mentioned, we give specific names, $\mathbf{w}$, 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 $\mathbf{w}$.

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 $M N$ 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 $\mathtt{fetch}[\mathbf{w}']M$ at $\mathbf{w}$. To do so, we suspend our current work at $\mathbf{w}$ and begin a new continuation on $\mathbf{w}'$ to evaluate M. The bottom of this continuation will be $\mathtt{return}\,\mathbf{w}$, which awaits a value to return to our old continuation at $\mathbf{w}$.

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 $\mathtt{return}\,\mathbf{w}$ or $\mathtt{finish}$ at its bottom. A continuation stack C is simply a list of pending continuations. The continuation $\mathtt{finish}$ 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 $\mathbf{w}$ is where the computation is taking place. The world $\mathbf{w}$ 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 figure 3, 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 in figure 5. 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. Also in this figure are the definitions of $\Sigma \vdash \ell : A@\mathbf{w}$ (which simply ensures that $\mathbf{w}$'s entry in Σ maps ℓ to

$$
\begin{array}{rcll}
\text{types} & A, B & ::= & \Box A \mid A \supset B \mid \Diamond B \\
\text{configs} & \mathbb{W} & ::= & \{\mathbf{w}_1 : \langle C_1, b_1 \rangle, \cdots\} \\
\text{networks} & \mathbb{N} & ::= & \mathbb{W}; \mathbf{w} : [k, M] \\
\text{tables} & b & ::= & \bullet \mid b, \ell = v \\
\text{config types} & \Sigma & ::= & \{\mathbf{w}_1 : \tau_1, \cdots, \mathbf{w}_i : \tau_i\} \\
\text{table types} & \tau & ::= & \bullet \mid \tau, \ell : A \\
\text{world exps} & \mathrm{w} & ::= & \mathbf{w} \mid \omega \\
\text{world vars} & \omega & & \text{world names} \quad \mathbf{w} \\
\text{labels} & \ell & & \text{value vars} \quad x, y
\end{array}
$$

$$
\begin{array}{rcll}
\text{values} & v & ::= & \lambda x.M \mid \texttt{box}\ \omega.M \mid \mathbf{w}.\ell \\
\text{cont stacks} & C & ::= & \star \mid C{::}k \\
\text{conts} & k & ::= & \texttt{return}\,\mathbf{w} \mid \texttt{finish} \mid k \lhd f \\
\text{frames} & f & ::= & \circ\ N \mid v\ \circ \mid \texttt{here}\,\circ \mid \texttt{unbox}\,\circ \\
 & & \mid & \texttt{letd}\,\omega.x = \circ\ \texttt{in}\,N \\
\text{exps} & M, N & ::= & v \mid MN \mid x \mid \ell \mid \texttt{fetch}[\mathrm{w}]M \\
 & & \mid & \texttt{here}\,M \mid \texttt{get}\,\langle \mathrm{w} \rangle M \\
 & & \mid & \texttt{unbox}\,M \mid \texttt{letd}\,\omega.x = M\,\texttt{in}\,N
\end{array}
$$

Figure 4: Syntax of Lambda 5 type system

$$
\frac{}{\Sigma; \Omega, \omega \vdash \omega}\ \text{wvar}
\qquad
\frac{1 \leq j \leq i}{\{\mathbf{w}_1 : \tau_1, \ldots, \mathbf{w}_i : \tau_i\}; \Omega \vdash \mathbf{w}_j}\ \text{wname}
$$

$$
\frac{}{\{\mathbf{w}_1 : \tau_1, \cdots, \mathbf{w}_j : (\bullet, \ldots, \ell : A, \ldots), \cdots, \mathbf{w}_i : \tau_i\} \vdash \ell : A @ \mathbf{w}_j}\ \text{label type}
$$

$$
\frac{\begin{array}{c}\Sigma; \cdot; \cdot \vdash v_{j1} : A_{j1} @ \mathbf{w}_j \quad \cdots \quad \Sigma; \cdot; \cdot \vdash v_{jm} : A_{jm} @ \mathbf{w}_j \\ \Sigma = \{\cdots, \mathbf{w}_j : (\bullet, \ell_{j1} : A_{j1}, \ldots, \ell_{jm} : A_{jm}), \cdots\}\end{array}}{\Sigma \vdash (\bullet, \ell_{j1} = v_{j1}, \ldots, \ell_{jm} = v_{jm}) @ \mathbf{w}_j}\ \text{table type}
$$

Figure 5: Auxiliary Judgments: World well-formedness, label typing, table typing

A) and 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 $\mathbf{w}$.

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 the Lambda 5 external language (figure 3) 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 6 and 7, 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 is the rule for $\texttt{return}\,\mathbf{w}$. This rule ensures that the continuation stack at $\mathbf{w}$ 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.)

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

Figure 7: Network typing

Finally, we have the network typing judgment (figure 7). 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

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

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

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

Figure 6: Extended expression and continuation typing rules

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 8.

Much of the dynamic semantics is standard for a continuation-based abstract machine. The reduction rule for `unbox` (*unbox-reduce*) instantiates the mobile code with the current world. When we encounter a label (*lookup*), we look it up in the current world's table and proceed with that value. To publish a value (*here-reduce*), 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` (*letd-reduce*) 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 $\mathbf{w}$ (*fetch-push*) means saving the current continuation at $\mathbf{w}$, and beginning a new continuation to evaluate M at $\mathbf{w}'$ with $\mathtt{return}\,\mathbf{w}$ at its bottom. The rule for `get` (*get-push*) is essentially the same. Reducing $\mathtt{return}\,\mathbf{w}$ (*return*) simply moves the value to $\mathbf{w}$, resuming with its outermost continuation.

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, which is proved in appendix A. We then give a comparison to a hypothetical system where the rules act non-locally.

4.4 Type Safety

Type safety is the conjunction of two properties, progress (theorem 6) and type preservation (theorem 7). 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 $\mathbb{W}; \mathbf{w} : [\mathtt{finish}, v]$.[4] 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 6 (Progress)
If $\mathcal{D} :: \Sigma \vdash \mathbb{N}$
then *either* $\mathbb{N}$ *is terminal* *or* $\exists \mathbb{N}'.\mathbb{N} \mapsto \mathbb{N}'$.

Theorem 7 (Preservation)
If $\mathcal{D} :: \Sigma \vdash \mathbb{N}$ *and* $\mathcal{E} :: \mathbb{N} \mapsto \mathbb{N}'$
then $\exists \Sigma', \mathcal{F}.\Sigma' \supseteq \Sigma$ *and* $\mathcal{F} :: \Sigma' \vdash \mathbb{N}'$.

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 given in appendix A.

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. 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.

4.5 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

[4] Again observe that we choose to not require continuations in the configuration $\mathbb{W}$ to be empty.

$$\frac{\Omega;\Gamma,x:A@\omega \vdash_{\mathrm{H}} M : A'@\omega}{\Omega;\Gamma \vdash_{\mathrm{H}} \lambda x.M : A \supset A'@\omega} \supset I \qquad \frac{\Omega;\Gamma \vdash_{\mathrm{H}} N : A'@\omega \quad \Omega;\Gamma \vdash_{\mathrm{H}} M : A' \supset A@\omega}{\Omega;\Gamma \vdash_{\mathrm{H}} MN : A@\omega} \supset E \qquad \frac{\omega \in \Omega}{\Omega;\Gamma,x:A@\omega,\Gamma' \vdash_{\mathrm{H}} x : A@\omega}\ \mathrm{hyp}$$

$$\frac{\Omega,\omega;\Gamma \vdash_{\mathrm{H}} M : \Box A@\omega'}{\Omega,\omega;\Gamma \vdash_{\mathrm{H}} \texttt{unboxfrom}[\omega']M : A@\omega} \Box E \qquad \frac{\omega' \text{ fresh} \quad \Omega,\omega';\Gamma \vdash_{\mathrm{H}} M : A@\omega' \quad \omega \in \Omega}{\Omega;\Gamma \vdash_{\mathrm{H}} \texttt{box}\ \omega'.M : \Box A@\omega} \Box I$$

$$\frac{\omega' \text{ fresh} \quad \Omega;\Gamma \vdash_{\mathrm{H}} M : \Diamond A@\omega'' \quad \Omega,\omega';\Gamma,x:A@\omega' \vdash_{\mathrm{H}} N : B@\omega}{\Omega;\Gamma \vdash_{\mathrm{H}} \texttt{letdfrom}\,\langle\omega''\rangle\ \omega'.x = M\ \texttt{in}\,N : B@\omega} \Diamond E \qquad \frac{\Omega,\omega';\Gamma \vdash_{\mathrm{H}} M : A@\omega}{\Omega,\omega';\Gamma \vdash_{\mathrm{H}} \texttt{there}\,\langle\omega\rangle M : \Diamond A@\omega'} \Diamond I$$

Figure 9: Hypothetical system H5

Simpson's system $\mathbf{N}_{\Box\Diamond}$ [15]). It shares features with calculi discussed in section 6.

The typing rules for H5 are given in figure 9. H5 has no `get` or `fetch`; instead it replaces `here`, `unbox`, and `letd` with three new terms:

- `there` $\langle\omega\rangle M$, which computes M of type A at ω and then returns its address of type $\Diamond A$;
- `unboxfrom`$[\omega]M$, which computes M (of type $\Box A$) at ω, and then returns its value of type A;
- `letdfrom` $\langle\omega\rangle\ \omega'.y = M$ `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:

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

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!

5 Future Work

With the minimal set of connectives presented here, our system has the same theorems 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 $\bot$ and $\vee$, however, we must be careful. Compare the elimination rule for $\Box$ with the elimination for $\bot$ in Simpson's IS5:

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

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 $\bot$ 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 $\bot$ or $\vee$ we must make the slightly weaker claim that IS5 and Lambda 5 have the same theorems under assumptions about a single class only. This includes all theorems of the form $\omega; \cdot \vdash A@\omega$ because all worlds introduced in the proof of $A@\omega$ will be interaccessible with ω.

Because $\bot$ and $\vee$ reason non-locally, we require special considerations in the operational semantics. Falsehood is simple: since there is no value of type $\bot E$ we can initiate a remote procedure call which is known

app-push
$$\mathbb{W}; \mathbf{w} : [k, MN] \;\mapsto \mathbb{W}; \mathbf{w} : [k \lhd \circ\; N; M]$$

app-flip
$$\mathbb{W}; \mathbf{w} : [k \lhd \circ\; N; v] \;\mapsto \mathbb{W}; \mathbf{w} : [k \lhd v\; \circ, N]$$

app-reduce
$$\mathbb{W}; \mathbf{w} : [k \lhd (\lambda x.M)\circ, v] \;\mapsto \mathbb{W}; \mathbf{w} : [k, [v/x]M]$$

here-push
$$\mathbb{W}; \mathbf{w} : [k, \mathtt{here}\, M] \;\mapsto \mathbb{W}; \mathbf{w} : [k \lhd \mathtt{here}\circ, M]$$

unbox-push
$$\mathbb{W}; \mathbf{w} : [k, \mathtt{unbox}\, M] \;\mapsto \mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox}\circ, M]$$

return
$$\{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}' : [\mathtt{return}\, \mathbf{w}, v] \;\mapsto$$
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, v]$$

fetch-push
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathtt{fetch}[\mathbf{w}']M] \;\mapsto$$
$$\{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}' : [\mathtt{return}\, \mathbf{w}, M]$$

get-push
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathtt{get}\,\langle \mathbf{w}'\rangle M] \;\mapsto$$
$$\{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}' : [\mathtt{return}\, \mathbf{w}, M]$$

here-reduce
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k \lhd \mathtt{here}\circ, v] \;\mapsto$$
$$\{\mathbf{w} : \langle C; b, \ell = v\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathbf{w}.\ell] \quad (\ell \text{ fresh})$$

unbox-reduce
$$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox}\circ, \mathtt{box}\; \omega.M] \;\mapsto$$
$$\mathbb{W}; \mathbf{w} : [k, [\mathbf{w}/\omega]M]$$

lookup
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \ell] \;\mapsto$$
$$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, v] \qquad (b(\ell) = v)$$

letd-push
$$\mathbb{W}; \mathbf{w} : [k, \mathtt{letd}\, \omega.x = M\; \mathtt{in}\, N] \;\mapsto$$
$$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\, \omega.x = \circ\; \mathtt{in}\, N, M]$$

letd-reduce
$$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\, \omega.x = \circ\; \mathtt{in}\, N, \mathbf{w}'.\ell] \;\mapsto$$
$$\mathbb{W}; \mathbf{w} : [k, [\ell/x][\mathbf{w}'/\omega]N]$$

Figure 8: Dynamic Semantics

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.

Other future work includes incorporating recursion and other type constructs for functional programming. We also wish to implement such an extension as a real programming language for the ConCert project [5]. Therefore we need to consider lower-level details of an implementation such as distributed garbage collection, failure recovery, 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[5] 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] were perhaps the first to devise a 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. [6] 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 [13] 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 [11]. 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 cal-

[5] by way of compilation into shell scripts!

culus 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 explicitly code motion as in the *fetch* and *get* constructs. In particular, due to the lack of symmetry it is not possible to go back to a source world after a potentially remote procedure call except by returning a value.

Jia and Walker [7] give a judgmental account of an S5-like system based on hybrid logics, but do not compare it to known logics. Hybrid logics internalize worlds inside propositions by including a *proposition* that a value of type A resides a world ω, which we might write "$A \,\text{at}\, \omega$." 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 4.5. Like Moody, they give their network semantics as a process calculus with passive synchronization across processes as a primitive notion. As a result, they also must have multiple processes running at a particular location, each one associated with a label. 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] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless grid computing in conCert. In M. Parashar, editor, *Grid Computing – Grid 2002 Third International Workshop*, pages 112–125, Berlin, November 2002. Springer-Verlag.

[6] 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.

[7] Limin Jia and David Walker. Modal proofs as distributed programs. Technical Report TR-671-03, Princeton University, August 2003.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

APPENDIX

A Type Safety

In this appendix we provide proofs of type safety. We require a few straightforward lemmas before continuing, whose easy verification we omit. We have the principles of weakening and contraction, and substitution for truth and world assumptions, as stated earlier. We also admit weakening of the configuration type Σ (addition of fresh labels to the table types) for the table typing judgment $\Sigma \vdash b@\mathbf{w}$, expression typing judgment $\Sigma; \Omega; \Gamma \vdash M : A@\mathbf{w}$, and continuation typing judgment $\Sigma \vdash \mathbb{W}; k : A@\mathbf{w}$. For weakening of table typing, we can only add labels to worlds other than $\mathbf{w}$. Because our expression and continuation typing rules are syntax directed, we are able to apply inversion to them. Finally, we have the standard *canonical forms* lemma.

Lemma 1 (Canonical Forms) *The forms of closed values are predicted by their types.*

	If $\mathcal{D}$ ::	*Then* $v = \dots$
1.	$\Sigma; \cdot; \cdot \vdash v : \Box A@\mathrm{w}$	$\boldsymbol{box}\ \omega.M$
2.	$\Sigma; \cdot; \cdot \vdash v : \Diamond A@\mathrm{w}$	$\boldsymbol{w}.\ell$
3.	$\Sigma; \cdot; \cdot \vdash v : A \supset B@\mathrm{w}$	$\lambda x.M$

Proof is by induction on the typing derivation $\mathcal{D}$.

Theorem 6 (Progress)

If $\quad \mathcal{D} :: \Sigma \vdash \mathbb{N}$
then $\quad$ either $\mathbb{N}$ is terminal $\quad$ or $\quad \exists \mathbb{N}'.\mathbb{N} \mapsto \mathbb{N}'$.

Proof is by induction on the derivation $\mathcal{D}$. For each case, say $\mathbb{N} = \mathbb{W}; \mathbf{w} : [k, M]$, where $\mathbb{W} = \{\mathbf{w}_1 : \langle C_1, b_1\rangle, \cdots, \mathbf{w}_i : \langle C_i, b_i\rangle\}$. By inversion on $\mathcal{D}$ we know that $\mathrm{dom}(\Sigma) = \mathrm{dom}(\mathbb{W})$ and $\mathbf{w} \in \mathrm{dom}(\Sigma)$. We also know there exists some type A such that:

$$\begin{array}{lcll} \mathcal{B}_i & :: & \Sigma \vdash b_i@\mathbf{w}_i & (i \in 1 \dots n) \\ \mathcal{T} & :: & \Sigma; \cdot; \cdot \vdash M : A@\mathbf{w} & \\ \mathcal{C} & :: & \Sigma \vdash \mathbb{W}; k : A@\mathbf{w} & \end{array}$$

We start with cases on M.

$M = \dots$	**case**
$M'N'$	$\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k \lhd \circ N', M']$ by app-push.
x	Impossible, as no rule can conclude $\Sigma; \cdot; \cdot \vdash x : A@\mathbf{w}$.
ℓ	$\mathbf{w} = \mathbf{w}_j$ for some j. By inversion on $\mathcal{T}$ we have $\mathcal{T}_1 :: \Sigma \vdash \ell : A@\mathbf{w}_j$, therefore $\ell : A \in \Sigma$. Therefore by inversion on $\mathcal{B}_j$ we have that $b_j = (\bullet, \dots, \ell = v, \dots)$. So by lookup we have $\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k, v]$.
$\mathtt{here}\, M'$	$\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k \lhd \mathtt{here} \circ, M']$ by here-push.
$\mathtt{get}\, \langle \mathrm{w} \rangle M'$	$\mathbb{W} = \{\mathbf{w} : \langle C, b\rangle; \mathbf{w}_s\}$. By inversion on $\mathcal{T}$ we see that $\Sigma; \cdot \vdash \mathrm{w}$. Since Ω is empty, we know $\mathrm{w} = \mathbf{w}_j$ for some $1 \le j \le i$. Therefore $\mathbb{N}' = \{\mathbf{w} : \langle C :: k, b\rangle; \mathbf{w}_s\}; \mathbf{w}_j : [\mathtt{return}\, \mathbf{w}, M']$ by get-push.

$\texttt{fetch}[\mathrm{w}]M'$	As for $\texttt{get}$, we get by inversion on $\mathcal{T}$ that $\mathrm{w} = \mathbf{w}_j$ for some $1 \le j \le i$. Therefore $\mathbb{N}' = \{\mathbf{w} : \langle C :: k, b\rangle; \mathbf{w}_s\}; \mathbf{w}_j : [\texttt{return}\,\mathbf{w}, M']$ by fetch-push.
$\texttt{unbox}\,M'$	$\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k \lhd \texttt{unbox}\,\circ, M']$ by unbox-push.
$\texttt{letd}$ $x\langle\mathrm{w}\rangle = M'$ $\texttt{in}\,N$	$\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k \lhd \texttt{letd}\,x\langle\mathrm{w}\rangle = \circ\,\texttt{in}\,N, M']$ by letd-push.
v	If M is a value v, then we proceed by cases on the continuation k.

$k = \ldots$	**case**
$\texttt{finish}$	$\mathbb{N}$ is terminal.
$\texttt{return}\,\mathbf{w}'$	By inversion on $\mathcal{T}$ we know that $\mathbf{w}' = \mathbf{w}_j$ for $1 \le j \le i$ and $\mathbb{W} = \{\mathbf{w}_j : \langle C_j :: k_j, b_j\rangle; \mathbf{w}_s\}$. So $\mathbb{N}' = \{\mathbf{w}_j : \langle C_j, b_j\rangle; \mathbf{w}_s\}; \mathbf{w}_j : [k_j, v]$ by return.
$k' \lhd f$	If k is a stack of frames then we proceed by cases on the top frame f.

$f = \ldots$	**case**
$\circ\, N$	By app-flip, $\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k' \lhd v\,\circ, N]$.
$v'\,\circ$	By inversion on $\mathcal{C}$, $\Sigma; \cdot; \cdot \vdash v' : A \supset B@\mathbf{w}$. Therefore by canonical forms (lemma 1), $v' = \lambda x.M'$. Then $\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k', [v/x]M']$ by app-reduce. Observe that substitution is defined for all v, x, M'.
$\texttt{here}\,\circ$	$\mathbb{W} = \{\mathbf{w} : \langle C, b\rangle; \mathbf{w}_s\}$ so by here-reduce, $\mathbb{N}' = \{\mathbf{w} : \langle C, (b, \ell = v)\rangle; \mathbf{w}_s\}; \mathbf{w} : [k', \mathbf{w}.\ell]$ where ℓ is any fresh label.
$\texttt{unbox}\,\circ$	Only one rule applies for $\mathcal{C}$ so we know that $A = \Box A'$. By canonical forms (lemma 1) on $\mathcal{T}$, $v = \texttt{box}\ \omega.M'$. Therefore $\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k', [\mathbf{w}/\omega]M']$ by unbox-reduce. Note that substitution is defined for all $\mathbf{w}, \omega, M'$.
$\texttt{letd}$ $x\langle\omega\rangle = \circ$ $\texttt{in}\,N$	Only one rule applies for $\mathcal{C}$ so we know that $A = \Diamond A'$. By canonical forms (lemma 1) on $\mathcal{T}$, $v = \mathbf{w}'.\ell$. Therefore $\mathbb{N}' = \mathbb{W}; \mathbf{w} : [k', [\ell/x][\mathbf{w}'/\omega]N]$ by letd-reduce. Note again that substitution is defined everywhere.

This exhausts the cases, completing the proof. $\Box$

Theorem 7 (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 is by induction on the derivation $\mathcal{E}$. For each case say $\mathbb{N} = \mathbb{W}; \mathbf{w} : [k, M]$. As in theorem 6, by inversion on $\mathcal{D}$ we know $\mathrm{dom}(\Sigma) = \mathrm{dom}(\mathbb{W})$ and $\mathbf{w} \in \mathrm{dom}(\Sigma)$. Without loss of generality call these worlds $\mathbf{w}_1 \ldots \mathbf{w}_n$. We also know there exists some type A such that:

$$\begin{array}{lcll} \mathcal{B}_i & :: & \Sigma \vdash b_i@\mathbf{w}_i & (i \in 1 \ldots n) \\ \mathcal{T} & :: & \Sigma; \cdot; \cdot \vdash M : A@\mathbf{w} & \\ \mathcal{C} & :: & \Sigma \vdash \mathbb{W}; k : A@\mathbf{w} & \end{array}$$

And we must show a corresponding Σ', A', $\mathcal{B}'_i$, $\mathcal{T}'$, and $\mathcal{C}'$ such that $\Sigma' \vdash \mathbb{N}'$.

Except in the case where we reduce `here` and update the tables with a new label the configuration typing and table types remain the same. Therefore $\Sigma' = \Sigma$ and $\mathcal{B}'_i = \mathcal{B}_i$ except where noted.

$\{\mathbf{w}_j : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [\mathtt{return}\,\mathbf{w}_j, \mathtt{box}\ \omega.M] \quad \mapsto$
$\{\mathbf{w}_j : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}_j : [k, \mathtt{box}\ \omega.M]$

By inversion on $\mathcal{C}$, we know $\mathcal{C}_0 :: \Sigma \vdash \{\mathbf{w}_j : \langle C, b\rangle; \mathrm{w}_s\}; k : A@\mathbf{w}_j$. Then, we have $\mathcal{C}' = \mathcal{C}_0$.
By inversion on $\mathcal{T}$, we have $\Sigma; \cdot; \cdot \vdash M : A@\omega$, so by the box rule we have $\Sigma; \cdot; \cdot \vdash \mathtt{box}\ \omega.M : \Box A@\mathbf{w}_j$ as required.

$\{\mathbf{w}_j : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [\mathtt{return}\,\mathbf{w}_j, \mathbf{w}'.\ell] \quad \mapsto$
$\{\mathbf{w}_j : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w}_j : [k, \mathbf{w}'.\ell]$

By inversion on $\mathcal{C}$, we know $\mathcal{C}_0 :: \Sigma \vdash \{\mathbf{w}_j : \langle C, b\rangle; \mathrm{w}_s\}; k : A@\mathbf{w}_j$. Then, we have $\mathcal{C}' = \mathcal{C}_0$.
By inversion on $\mathcal{T}$, we have $\Sigma \vdash \ell : A@\mathbf{w}'$, so by the dia rule we have $\Sigma; \cdot; \cdot \vdash \mathbf{w}'.\ell : \Diamond A@\mathbf{w}_j$ as required.

$\mathbb{W}; \mathbf{w} : [k \lhd \circ\, N; v] \quad \mapsto$
$\mathbb{W}; \mathbf{w} : [k \lhd v \circ, N]$

By inversion on $\mathcal{C}$, $A = A'' \supset A'$ and we have $\mathcal{C}_1 :: \Sigma \vdash \mathbb{W}; k : A'@\mathbf{w}$ and $\mathcal{C}_2 :: \Sigma; \cdot; \cdot \vdash N : A''@\mathbf{w}$.
Then, $\mathcal{T}' = \mathcal{C}_2$, and $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}_1\\ \vdots \\ \Sigma \vdash \mathbb{W}; k : A'@\mathbf{w}\end{array} \quad \begin{array}{c}\mathcal{T}\\ \vdots \\ \Sigma; \cdot; \cdot \vdash v : A'' \supset A'@\mathbf{w}\end{array}}{\Sigma \vdash \mathbb{W}; k \lhd v \circ : A''@\mathbf{w}}$$

$\mathbb{W}; \mathbf{w} : [k, MN] \quad \mapsto$
$\mathbb{W}; \mathbf{w} : [k \lhd \circ N; M]$

By inversion on $\mathcal{T}$, we have $\mathcal{T}_1 :: \Sigma; \cdot; \cdot \vdash M : A'' \supset A'@\mathbf{w}$ and $\mathcal{T}_2 :: \Sigma; \cdot; \cdot \vdash N : A''@\mathbf{w}$.
Then $\mathcal{T}' = \mathcal{T}_1$ and $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}\\ \vdots \\ \Sigma \vdash \mathbb{W}; k : A'@\mathbf{w}\end{array} \quad \begin{array}{c}\mathcal{T}_2\\ \vdots \\ \Sigma; \cdot; \cdot \vdash N : A''@\mathbf{w}\end{array}}{\Sigma \vdash \mathbb{W}; k \lhd \circ N : A'' \supset A'@\mathbf{w}}$$

$\mathbb{W}; \mathbf{w} : [k \lhd (\lambda x.M)\circ, v] \quad \mapsto$
$\mathbb{W}; \mathbf{w} : [k, [v/x]M]$

By inversion on $\mathcal{C}$ we have $\mathcal{C}_1 :: \Sigma \vdash \mathbb{W}; k : A'@\mathbf{w}$ and $\mathcal{C}_2 :: \Sigma; \cdot; \cdot \vdash \lambda x.M : A \supset A'@\mathbf{w}$.
By inversion again on $\mathcal{C}_2$ we have $\mathcal{C}_3 :: \Sigma; \cdot; x : A@\mathbf{w} \vdash M : A'@\mathbf{w}$.
Therefore $\mathcal{C}' = \mathcal{C}_1$. By substitution on $\mathcal{C}_3$ and $\mathcal{T}$, we have $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash [v/x]M : A'@\mathbf{w}$, as required.

$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathtt{fetch}[\mathbf{w}']M]$ $\mapsto$
$\{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}' : [\mathtt{return}\,\mathbf{w}, M]$

By inversion on $\mathcal{T}$, $\mathbf{w}' \in \mathbb{W}$ and $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash M : \Box A@\mathbf{w}'$.
Thus $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}\\ \vdots\\ \Sigma \vdash \{\mathbf{w} : \langle C; b\rangle; \mathrm{w}_s\}; k : \Box A@\mathbf{w}\end{array}}{\Sigma \vdash \{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathtt{return}\,\mathbf{w} : \Box A@\mathbf{w}'}.$$

$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathtt{get}\,\langle \mathbf{w}'\rangle M]$ $\mapsto$
$\{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathbf{w}' : [\mathtt{return}\,\mathbf{w}, M]$

Similarly, by inversion on $\mathcal{T}$, $\mathbf{w}' \in \mathbb{W}$ and $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash M : \Diamond A@\mathbf{w}'$.
Thus $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}\\ \vdots\\ \Sigma \vdash \{\mathbf{w} : \langle C; b\rangle; \mathrm{w}_s\}; k : \Diamond A@\mathbf{w}\end{array}}{\Sigma \vdash \{\mathbf{w} : \langle C{::}k, b\rangle; \mathrm{w}_s\}; \mathtt{return}\,\mathbf{w} : \Diamond A@\mathbf{w}'}.$$

$\mathbb{W}; \mathbf{w} : [k, \mathtt{here}\,M]$ $\mapsto$
$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{here}\,\circ, M]$

By inversion on $\mathcal{T}$ we have $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash M : A'@\mathbf{w}$.
Then $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}\\ \vdots\\ \Sigma \vdash \mathbb{W}; k : \Diamond A'@\mathbf{w}\end{array}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{here}\,\circ : A'@\mathbf{w}}$$

$\mathbb{W}; \mathbf{w} : [k, \mathtt{unbox}\,M]$ $\mapsto$
$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox}\,\circ, M]$

By inversion on $\mathcal{T}$ we have $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash M : \Box A'@\mathbf{w}$.
Then $\mathcal{C}' =$

$$\frac{\begin{array}{c}\mathcal{C}\\ \vdots\\ \Sigma \vdash \mathbb{W}; k : A'@\mathbf{w}\end{array}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{unbox}\,\circ : \Box A'@\mathbf{w}}$$

$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox}\,\circ, \mathtt{box}\ \omega.M]$ $\mapsto$
$\mathbb{W}; \mathbf{w} : [k, [\mathbf{w}/\omega]M]$

By inversion on $\mathcal{C}$ we have $\mathcal{C}' :: \Sigma \vdash \mathbb{W}; k : A@\mathbf{w}$.
By inversion on $\mathcal{T}$ we have $\mathcal{T}_1 :: \Sigma; \omega; \cdot \vdash M : A@\omega$. By world substitution, we have $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash [\mathbf{w}/\omega]M : A@\mathbf{w} = [\mathbf{w}/\omega]\mathcal{T}_1$, as required.

$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \ell]$ $\mapsto$
$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, v]$
$(b(\ell) = v)$

$\mathbf{w} = \mathbf{w}_j$ for some j.
$\mathcal{C}' = \mathcal{C}$.
Because $b(\ell) = v$, by inversion on $\mathcal{B}_j$ we have $\Sigma; \cdot; \cdot \vdash v : B@\mathbf{w}_j$.
By inversion on $\mathcal{T}$ we have $\mathcal{T}_1 :: \Sigma \vdash \ell : A@\mathbf{w}_j$. Because the domain of Σ is disjoint, we know that $A = B$ so $\mathcal{T}' = \mathcal{T}_1$.

$\mathbb{W}; \mathbf{w} : [k, \mathtt{letd}\, x\langle\omega\rangle = M \,\mathtt{in}\, N] \quad \mapsto$
$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\, x\langle\omega\rangle = \circ \,\mathtt{in}\, N, M]$

By inversion on $\mathcal{T}$, there exists B such that $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash M : \Diamond B@\mathbf{w}$ and $\mathcal{T}_1 :: \Sigma; \omega'; x : B@\omega' \vdash N : A@\mathbf{w}$.
Then $\mathcal{C}' =$

$$\dfrac{\begin{array}{c}\mathcal{C}\\ \vdots \\ \Sigma \vdash \mathbb{W}; k : A@\mathbf{w}\end{array} \quad \begin{array}{c}\mathcal{T}_1\\ \vdots \\ \Sigma; \omega'; x : B@\omega' \vdash N : A@\mathbf{w}\end{array}}{\Sigma \vdash \mathbb{W}; k \lhd \mathtt{letd}\, x\langle\omega\rangle = \circ \,\mathtt{in}\, N : \Diamond B@\omega}$$

$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\, x\langle\omega\rangle = \circ \,\mathtt{in}\, N, \mathbf{w}'.\ell] \quad \mapsto$
$\mathbb{W}; \mathbf{w} : [k, [\ell/x][\mathbf{w}'/\omega]N]$

By inversion on $\mathcal{C}$, we have $\mathcal{C}' :: \Sigma \vdash \mathbb{W}; k : B@\mathbf{w}$ and $\mathcal{C}_1 :: \Sigma; \omega'; x : A@\omega' \vdash N : B@\mathbf{w}$. By world substitution, we have $\mathcal{C}_2 :: \Sigma; \cdot; x : A@\mathbf{w}' \vdash [\mathbf{w}'/\omega']N : B@\mathbf{w} = [\mathbf{w}'/\omega]\mathcal{C}_1$.
By inversion on $\mathcal{T}$, we have $\mathcal{T}_1 :: \Sigma \vdash \ell : A@\mathbf{w}'$. Therefore by the lab rule we have $\mathcal{T}_2 :: \Sigma; \cdot; \cdot \vdash \ell : A@\mathbf{w}'$.
By substitution on $\mathcal{C}_2$ and $\mathcal{T}_2$ we have $\mathcal{T}' :: \Sigma; \cdot; \cdot \vdash [\ell/x][\mathbf{w}'/\omega]N : B@\mathbf{w}$, as required.

$\{\mathbf{w} : \langle C, b\rangle; \mathrm{w}_s\}; \mathbf{w} : [k \lhd \mathtt{here}\, \circ, v] \quad \mapsto$
$\{\mathbf{w} : \langle C; b, \ell = v\rangle; \mathrm{w}_s\}; \mathbf{w} : [k, \mathbf{w}.\ell]$
(ℓ fresh)

In this case S' and $\mathcal{B}'_i$ will differ from the inputs.
$\mathbf{w} = \mathbf{w}_j$ for some j.
By inversion on $\mathcal{C}$, we have $\mathcal{C}_1 :: \Sigma \vdash \mathbb{W}; k : \Diamond A@\mathbf{w}_j$.
$\Sigma = \{\mathbf{w}_1 : \tau_1, \cdots, \mathbf{w}_i : \tau_i\}$.
Let $\Sigma' = \{\mathbf{w}_1 : \tau_1, \cdots \mathbf{w}_j : (\tau_j, \ell : A) \cdots, \mathbf{w}_i : \tau_i\}$. Note that because ℓ is fresh, $\Sigma' \supseteq \Sigma$.
Then $\mathcal{C}' = \text{weaken-}\Sigma(\mathcal{C}_1)$.
$\mathcal{T}' =$

$$\dfrac{\dfrac{}{\{\cdots \mathbf{w}_j : (\tau_j, \ell : A) \cdots\} \vdash \ell : A@\mathbf{w}_j}}{\{\cdots \mathbf{w}_j : (\tau_j, \ell : A) \cdots\}; \cdot; \cdot \vdash \mathbf{w}_j.\ell : \Diamond A@\mathbf{w}_j}$$

By inversion on $\mathcal{B}_j$ we get $\mathcal{T}_{j1} :: \Sigma; \cdot; \cdot \vdash v_{j1} : A_{j1}@\mathbf{w}_j$ through $\mathcal{T}_{jm} :: \Sigma; \cdot; \cdot \vdash v_{jm} : A_{jm}@\mathbf{w}_j$.
$\mathcal{B}'_i = \text{weaken-}\Sigma(\mathcal{B}_i)$ for $i \neq j$, and $\mathcal{B}'_j =$

$$\dfrac{\begin{array}{c}\mathcal{T}_{j1}\\ \vdots \\ \Sigma'; \cdot; \cdot \vdash v_{j1} : A_{j1}@\mathbf{w}_j\end{array} \quad \cdots \quad \begin{array}{c}\mathcal{T}_{jm}\\ \vdots \\ \Sigma'; \cdot; \cdot \vdash v_{jm} : A_{jm}@\mathbf{w}_j\end{array} \quad \begin{array}{c}\text{weaken-}\Sigma(\mathcal{T})\\ \vdots \\ \Sigma'; \cdot; \cdot \vdash v : A@\mathbf{w}_j\end{array}}{\Sigma \vdash' @(b_j, \ell = v)\mathbf{w}_j}$$

This completes the proof of preservation. $\Box$

B Machine Checkable Proofs

This appendix contains the Twelf code for the proofs of theorem 3 (cut), theorem 5 (equivalence to sequent calculus), and normalization. Also included are simplified versions of type safety. The code is available electronically from `http://www.cs.cmu.edu/~concert/` and each proof is verifiable by Twelf's metatheorem checker.

B.1 Cut, Equivalence, Normalization

```
world : type.                 %name world W o.
prop : type.                  %name prop A.

%%% Natural deduction
```

```
@ : prop -> world -> type.      %name @ N.
%infix none 1 @.

% Implication
=> : prop -> prop -> prop.
%infix right 8 =>.
=>I : (A @ W -> B @ W) -> (A => B @ W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Necessity
! : prop -> prop.
%prefix 9 !.
!I : ({o:world} A @ o) -> ! A @ W.
!E : ! A @ W -> A @ W.
!G : ! A @ W' -> ! A @ W.

% Possibility
? : prop -> prop.
%prefix 9 ?.
?I : A @ W -> ? A @ W.
?E : ? A @ W -> ({o:world} A @ o -> C @ W) -> C @ W.
?G : ? A @ W' -> ? A @ W.

%%% Sequent calculus (SS5)

hyp : prop -> world -> type.        %name hyp H h.
conc : prop -> world -> type.       %name conc D.

% Judgmental rules
init : hyp A W -> conc A W.

% Implication
=>R : (hyp A W -> conc B W)
       -> conc (A => B) W.
=>L : conc A W
       -> (hyp B W -> conc C U)
       -> (hyp (A => B) W -> conc C U).

% Necessity
!R : ({o:world} conc A o)
      -> conc (! A) W.
!L : (hyp A W' -> conc C U)
      -> (hyp (! A) W -> conc C U).

% Possibility
?R : conc A W'
      -> conc (? A) W.
?L : ({o:world} hyp A o -> conc C U)
      -> (hyp (? A) W -> conc C U).

%%% Admissibility of Cut

cut : {A:prop} conc A W -> (hyp A W -> conc C U) -> conc C U -> type.
%mode cut +A +D +E -F.

% Initial cuts
ci_l : cut A (init H) ([h] E h) (E H).
ci_r : cut A D ([h] init h) D.
```

```
% Principal cuts
c_=> : cut (A1 => A2) (=>R ([h1] D2 h1)) ([h] =>L (E1 h) ([h2] E2 h h2) h) F
    <- cut (A1 => A2) (=>R ([h1] D2 h1)) ([h] E1 h) E1'
    <- ({h2:hyp A2 W} cut (A1 => A2) (=>R ([h1] D2 h1)) ([h] E2 h h2) (E2' h2))
    <- cut A1 E1' ([h1] D2 h1) F1
    <- cut A2 F1 ([h2] E2' h2) F.
c_! : cut (! A1) (!R ([o] D1 o)) ([h] !L ([h1] E1 h h1) h) F
       <- ({h1:hyp A1 W'} cut (! A1) (!R ([o] D1 o)) ([h] E1 h h1) (E1' h1))
       <- cut A1 (D1 W') ([h1] E1' h1) F.
c_? : cut (? A1) (?R D1) ([h] ?L ([o][h1] E1 h o h1) h) F
       <- ({o:world} {h1:hyp A1 o} cut (? A1) (?R D1) ([h] E1 h o h1) (E1' o h1))
       <- cut A1 D1 ([h1] E1' W' h1) F.

% Right commuting cuts
cr_init : cut A D ([h] init H) (init H).
cr_=>R : cut A D ([h] =>R ([h1] E1 h h1)) (=>R ([h1] F1 h1))
      <- ({h1:hyp C1 U} cut A D ([h] E1 h h1) (F1 h1)).
cr_=>L : cut A D ([h] =>L (E1 h) ([h2] E2 h h2) H) (=>L F1 ([h2] F2 h2) H)
      <- cut A D ([h] E1 h) F1
      <- ({h2:hyp B2 U'} cut A D ([h] E2 h h2) (F2 h2)).
cr_!R : cut A D ([h] !R ([o] E1 h o)) (!R ([o] F1 o))
     <- ({o:world} cut A D ([h] E1 h o) (F1 o)).
cr_!L : cut A D ([h] !L ([h1] E1 h h1) H) (!L ([h1] F1 h1) H)
     <- ({h1:hyp B1 U'} cut A D ([h] E1 h h1) (F1 h1)).
cr_?R : cut A D ([h] ?R (E1 h)) (?R F1)
     <- cut A D ([h] E1 h) F1.
cr_?L : cut A D ([h] ?L ([o][h1] E1 o h1 h) H) (?L ([o] [h1] F1 o h1) H)
     <- ({o:world} {h1:hyp B1 o} cut A D ([h] E1 o h1 h) (F1 o h1)).

% Left commuting cuts
cl_=>L : cut A (=>L D1 ([h2] D2 h2) H) ([h] E h) (=>L D1 ([h2] F2 h2) H)
      <- ({h2:hyp B2 U'} cut A (D2 h2) ([h] E h) (F2 h2)).
cl_!L : cut A (!L ([h1] D1 h1) H) ([h] E h) (!L ([h1] F1 h1) H)
     <- ({h1:hyp B1 U'} cut A (D1 h1) ([h] E h) (F1 h1)).
cl_?L : cut A (?L ([o][h1] D1 o h1) H) ([h] E h) (?L ([o][h1] F1 o h1) H)
     <- ({o:world} {h1:hyp B1 o} cut A (D1 o h1) ([h] E h) (F1 o h1)).

%block lo : block {o:world}.
%block lh : some {A:prop} {W:world} block {h:hyp A W}.
%worlds (lo | lh) (cut A D E F).
%total {A [D E]} (cut A D E F).

%%% Translation ND to SEQ

ndseq : A @ W -> conc A W -> type.  %name ndseq R r.
%mode ndseq +N -D.

% Implication
ns_=>I : ndseq (=>I ([u1] N2 u1)) (=>R ([h1] D2 h1))
      <- ({u1:A1 @ W} {h1:hyp A1 W}
        ndseq u1 (init h1) -> ndseq (N2 u1) (D2 h1)).
ns_=>E : ndseq (=>E N2 N1) D
      <- ndseq N2 D'
      <- ndseq N1 D1
      <- cut (A1 => A2) D' ([h] =>L D1 ([h2] init h2) h) D.
```

```
% Necessity
ns_!I : ndseq (!I ([o] N1 o)) (!R ([o] D1 o))
     <- ({o:world} ndseq (N1 o) (D1 o)).
ns_!E : ndseq (!E N1) D
     <- ndseq N1 D'
         <- cut (! A1) D' ([h] !L ([h1] init h1) h) D.
ns_!G : ndseq (!G N1) D
     <- ndseq N1 D'
         <- cut (! A1) D' ([h] !R ([o] !L ([ho] init ho) h)) D.

% Possibility
ns_?I : ndseq (?I N1) (?R D1)
     <- ndseq N1 D1.
ns_?E : ndseq (?E N1 ([o][u1] N2 o u1)) D
     <- ndseq N1 D1'
     <- ({o:world} {u1:A1 @ o} {h1:hyp A1 o}
           ndseq u1 (init h1) -> ndseq (N2 o u1) (D2 o h1))
     <- cut (? A1) D1' ([h] ?L ([o][h1] D2 o h1) h) D.
ns_?G : ndseq (?G N1) D
     <- ndseq N1 D'
     <- cut (? A1) D' ([h] ?L ([o] [h1] ?R (init h1)) h) D.

%block luhs : some {A:prop} {W:world}
           block {u:A @ W} {h:hyp A W} {r:ndseq u (init h)}.
%worlds (lo | luhs) (ndseq N D).
%total N (ndseq N D).

%%% Translation SEQ to ND

seqnd : conc A W -> A @ W -> type.  %name seqnd S.
hypnd : hyp A W -> A @ W -> type.   %name hypnd T t.
%mode seqnd +D -N.
%mode hypnd +H -N.

% Init
sn_init : seqnd (init H) D
       <- hypnd H D.

% Implication
sn_=>R : seqnd (=>R ([h1] D2 h1)) (=>I ([u1] N2 u1))
      <- ({h1:hyp A1 W} {u1:A1 @ W}
        hypnd h1 u1 -> seqnd (D2 h1) (N2 u1)).

sn_=>L : seqnd (=>L D1 ([h2] D2 h2) H) (N2 (=>E N N1))
      <- seqnd D1 N1
      <- ({h2:hyp A2 W} {u2:A2 @ W}
        hypnd h2 u2 -> seqnd (D2 h2) (N2 u2))
      <- hypnd H N.

% Necessity
sn_!R : seqnd (!R ([o] D1 o)) (!I ([o] N1 o))
     <- ({o:world} seqnd (D1 o) (N1 o)).
sn_!L : seqnd (!L ([h1] D2 h1) H) (N2 (!E (!G N)))
     <- ({h1:hyp A1 W'} {u1:A1 @ W'}
           hypnd h1 u1 -> seqnd (D2 h1) (N2 u1))
     <- hypnd H N.
```

```
% Possibility
sn_?R : seqnd (?R D1) (?G (?I N1))
     <- seqnd D1 N1.
sn_?L : seqnd (?L ([o][h1] D2 o h1) H) (?E (?G N) ([o] [u1] N2 o u1))
     <- ({o:world} {h1:hyp A1 o} {u1:A1 @ o}
           hypnd h1 u1 -> seqnd (D2 o h1) (N2 o u1))
     <- hypnd H N.

%block lhut : some {A:prop} {W:world}
           block {h:hyp A W} {u:A @ W} {t:hypnd h u}.
%worlds (lo | lhut) (seqnd D N) (hypnd H N').
%total (D H) (seqnd D N) (hypnd H N').

%%% Normal deductions
norm : A @ W -> type.           %name norm M.
elim : A @ W -> type.           %name elim E e.
%mode norm +N.
%mode elim +N.

% Coercion
n_elim : norm N <- elim N.

% Implication
n_=>I : norm (=>I ([u1] N2 u1))
     <- ({u1:A1 @ W} elim u1 -> norm (N2 u1)).
n_=>E : elim (=>E N2 N1)
     <- elim N2
     <- norm N1.

% Necessity
n_!I : norm (!I ([o] N1 o))
     <- ({o:world} norm (N1 o)).
% !E by itself is redundant, but could be admitted
%{
n_!E : elim (!E N1)
     <- elim N1.
}%
n_!E!G : elim (!E (!G N1))
     <- elim N1.

% Possibility
% ?I and ?E by themselves are redundant, but could be admitted
%{
n_?I : norm (?I N1)
    <- norm N1.
n_?E : norm (?E N ([o][u1] N2 o u1))
    <- elim N
    <- ({o:world} {u1:A1 @ o} elim u1 -> norm (N2 o u1)).
}%
n_?G?I : norm (?G (?I N1))
      <- norm N1.
n_?E?G : norm (?E (?G N) ([o] [u1] N2 o u1))
      <- elim N
      <- ({o:world} {u1:A1 @ o} elim u1 -> norm (N2 o u1)).
```

```
%%% Translation of cut-free sequent derivations
%%% yields normal deductions

tnorm : seqnd D N -> norm N -> type.
telim : hypnd H N -> elim N -> type.
%mode tnorm +S -M.
%mode telim +T -E.

tn_init : tnorm (sn_init T) (n_elim E)
       <- telim T E.
tn_=>R : tnorm (sn_=>R ([h1][u1][t1] S2 h1 u1 t1)) (n_=>I ([u1][e1] M2 u1 e1))
      <- ({h1:hyp A1 W} {u1:A1 @ W} {t1:hypnd h1 u1} {e1:elim u1}
        telim t1 e1 -> tnorm (S2 h1 u1 t1) (M2 u1 e1)).
tn_=>L : tnorm (sn_=>L T ([h2][u2][e2] S2 h2 u2 e2) S1)
      (M2 _ (n_=>E M1 E))
      <- tnorm S1 M1
      <- ({h2:hyp A2 W} {u2:A2 @ W} {t2:hypnd h2 u2} {e2:elim u2}
        telim t2 e2 -> tnorm (S2 h2 u2 t2) (M2 u2 e2))
      <- telim T E.
tn_!R : tnorm (sn_!R ([o] S1 o)) (n_!I ([o] M1 o))
     <- ({o:world} tnorm (S1 o) (M1 o)).
tn_!L : tnorm (sn_!L T ([h1][u1][t1] S2 h1 u1 t1))
     (M2 _ (n_!E!G E))
     <- ({h1:hyp A1 W'} {u1:A1 @ W'} {t1:hypnd h1 u1} {e1:elim u1}
           telim t1 e1 -> tnorm (S2 h1 u1 t1) (M2 u1 e1))
     <- telim T E.
tn_?R : tnorm (sn_?R S1) (n_?G?I M1)
     <- tnorm S1 M1.
tn_?L : tnorm (sn_?L T ([o][h1][u1][t1] S2 o h1 u1 t1))
     (n_?E?G ([o][u1][e1] M2 o u1 e1) E)
     <- ({o:world} {h1:hyp A1 o} {u1:A1 @ o} {t1:hypnd h1 u1} {e1:elim u1}
           telim t1 e1 -> tnorm (S2 o h1 u1 t1) (M2 o u1 e1))
     <- telim T E.

%block lhute : some {A:prop} {W:world}
        block {h:hyp A W} {u:A @ W} {t:hypnd h u} {e:elim u}
              {te:telim t e}.
%worlds (lo | lhute) (tnorm S M) (telim T E).
%total (S T) (tnorm S M) (telim T E).

%%% Global normalization corollary
%%% formulated here only for closed terms
%%% all lemmas work for open terms

normalize : A @ W -> A @ W -> type.
%mode normalize +N -N'.

nn0 : normalize N N'
       <- ndseq N D
       <- seqnd D N'.

%worlds () (normalize N N').
%total [] (normalize N N').

normalization : normalize N N' -> norm N' -> type.
%mode normalization +NN -M.
```

```
nz0 : normalization (nn0 S R) M
       <- tnorm S M.
%worlds () (normalization NN M).
%total [] (normalization N N').
```

B.2 Type Safety

The operational semantics for which we mechanize type safety differs from the one given in section 4.4 in two ways. First, instead of distributing the continuation explicitly into stacks located at different worlds, we have one global continuation, pieces of which belong to the different worlds. Second, instead of keeping a table at each world and looking up labels, we substitute values which are intrinsicially bound to a world. Both differences are superficial, so the proof of progress contains essentially the same information as the progress proof in appendix A. The proofs of the substitution property and preservation are obviated by the representation, since every term (and value) is intrinsically indexed with its type and world. Twelf type-checking can therefore guarantee the type preservation theorem.

```
%%% Version 1: straightforward, big-step
%%% Using intrinsically typed terms
%%% Avoids run-time artefacts
```

```
%%% Definitions for term syntax
lam = =>I.
app = =>E.
box = !I.
unbox = !E.
fetch = ([W'] !G : ! A @ W' -> ! A @ W).
here = ?I.
letd = ?E.
get = ([W'] ?G : ? A @ W' -> ? A @ W).
```

```
%%% Values
value : {W:world} A @ W -> type.    %name value Q.
%mode value +W +V.
val_lam : value W (lam [x] M x).
val_box : value W (fetch W' (box [o] M o)).
val_here : value W (get W' (here V'))
        <- value W' V'.
```

```
%%% Evaluation (big-step)
eval : {W:world} A @ W -> A @ W -> type. %name eval D.
%mode eval +W +M -V.
ev_lam : eval W (lam [x] M x) (lam [x] M x).
ev_app : eval W (app M1 M2) V
      <- eval W M1 (lam [x] M1' x)
      <- eval W M2 V2
      <- eval W (M1' V2) V.
ev_box : eval W (box [o] M o) (fetch W (box [o] M o)).
ev_unbox : eval W (unbox M1) V
        <- eval W M1 (fetch W' (box [o] M1' o))
        <- eval W (M1' W) V.
ev_fetch : eval W (fetch W' M') (fetch W'' (box [o] M'' o))
        <- eval W' M' (fetch W'' (box [o] M'' o)).
ev_here : eval W (here M1) (get W (here V1))
       <- eval W M1 V1.
ev_letd : eval W (letd M1 ([o] [x:A1 @ o] M2 o x)) V
        <- eval W M1 (get W' (here V1'))
        <- eval W (M2 W' V1') V.
ev_get : eval W (get W' M') (get W'' (here V''))
      <- eval W' M' (get W'' (here V'')).
%worlds () (eval W M V).
%covers eval +W +M -V.

%%% Value soundness
%%% (evaluation returns a value)
vs : eval W M V -> value W V -> type.
%mode vs +D -Q.
vs_lam : vs (ev_lam) (val_lam).
vs_app : vs (ev_app D1' D2 D1) Q
      <- vs D1' Q.
vs_box : vs (ev_box) (val_box).
vs_unbox : vs (ev_unbox D1' D1) Q
        <- vs D1' Q.
vs_fetch : vs (ev_fetch D') (val_box)
        <- vs D' (val_box).
vs_here : vs (ev_here D1) (val_here Q1)
       <- vs D1 Q1.
vs_letd : vs (ev_letd D2 D1) Q
       <- vs D2 Q.
vs_get : vs (ev_get D') (val_here Q'')
      <- vs D' (val_here Q'').
%worlds () (vs D Q).
%total D (vs D Q).

%%% Version 2
%%% Small-step semantics
%%% Values, frames, and continuations as run-time artefacts
%%% No tables (using substitution instead)
%%% Final answer type and world is not (yet) represented.

% Values
# : prop -> world -> type.      %name # V.
%infix none 1 #.
```

```
vlam : (A @ W -> B @ W) -> A => B # W.  % lam ([x] M x)
vbox : ({o:world} A @ o) -> ! A # W.    % fetch _ (box [o] M o)
vhere : {W':world} A # W' -> ? A # W.   % get W' (here V)

% Inclusion of values in expressions
val : A # W -> A @ W.

% Frames
% frame A W C W' maps values A # W to continuations C at W
frame : prop -> world -> prop -> world -> type. %name frame F.

app1 : A @ W -> frame (A => B) W B W.       % app1 M2 ~ [v1] app v1 M2
app2 : A => B # W -> frame A W B W.     % app2 V1 ~ [v2] app V1 v2
unbox1 : frame (! A) W A W.         % unbox1  ~ [v] unbox v
fetch1 : {W':world} frame (! A) W' (! A) W. % fetch1 W' ~ [v'] fetch W' v'
here1 : frame A W (? A) W.          % here1 ~ [v] here1 v
letd1 : ({o:world} A @ o -> C @ W) -> frame (? A) W C W.
                    % letd1 ([o][x] M2 o x) ~ [v1] letd v1 ([o] [x] M2 o x)
get1 : {W':world} frame (? A) W' (? A) W.   % get1 W' ~ [v'] get W' v'

% Continuations
cont : prop -> world -> type.       %name cont K.
finish : cont A W.
; : cont C W' -> frame A W C W' -> cont A W.
%infix none 1 ;.

% Instructions
inst : prop -> world -> type.       %name inst I.
ev : A @ W -> inst A W.
ret : A # W -> inst A W.

% States
state : prop -> world -> type.      %name state S.
st : {W:world} cont A W -> inst A W -> state A W.

% Single step
% Only fetch, fetch1 and get, get1 change current world
step : state A W -> state A' W' -> type.
%mode step +S -S'.

st_val :    step (st W (K) (ev (val V)))
                 (st W (K) (ret V)).

st_app :    step (st W (K) (ev (app M1 M2)))
                 (st W (K ; app1 M2) (ev M1)).
st_app1 :   step (st W (K ; app1 M2) (ret V1))
             (st W (K ; app2 V1) (ev M2)).
st_app2 :   step (st W (K ; app2 (vlam [x] M1' x)) (ret V2))
                 (st W (K) (ev (M1' (val V2)))).
st_lam :    step (st W (K) (ev (lam [x] M x)))
             (st W (K) (ret (vlam [x] M x))).
```

```
st_box :    step (st W (K) (ev (box [o] M o)))
             (st W (K) (ret (vbox [o] M o))).
st_unbox :  step (st W (K) (ev (unbox M)))
             (st W (K ; unbox1) (ev M)).
st_unbox1 : step (st W (K ; unbox1) (ret (vbox [o] M o)))
             (st W (K) (ev (M W))).
st_fetch :  step (st W (K) (ev (fetch W' M')))
             (st W' (K ; fetch1 W') (ev M')).
st_fetch1 : step (st W' (K ; fetch1 W') (ret (vbox [o] M' o)))
             (st W (K) (ret (vbox [o] M' o))).

st_here :   step (st W (K) (ev (here M)))
             (st W (K ; here1) (ev M)).
st_here1 :  step (st W (K ; here1) (ret V))
             (st W (K) (ret (vhere W V))).
st_letd :   step (st W (K) (ev (letd M1 ([o] [x] M2 o x))))
             (st W (K ; letd1 ([o] [x] M2 o x)) (ev M1)).
st_letd1 :  step (st W (K ; letd1 ([o] [x] M2 o x)) (ret (vhere W' V1')))
             (st W (K) (ev (M2 W' (val V1')))).
st_get :    step (st W (K) (ev (get W' M)))
             (st W' (K ; get1 W') (ev M)).
st_get1 :   step (st W' (K ; get1 W') (ret (vhere W'' V)))
             (st W (K) (ret (vhere W'' V))).

% Last case to verify progress
st_halt :   step (st W (finish) (ret V))
             (st W (finish) (ret V)).

% Allow world parameters, otherwise totality is trivial
%worlds (lo) (step S S').
%total [] (step S S').
```