Modal Proofs As Distributed Programs*
(Extended Abstract)

Limin Jia David Walker

Princeton University
{ljia, dpw}@cs.princeton.edu
fax: 609-258-1771

Abstract. We develop a new foundation for distributed programming languages by defining an intuitionistic, modal logic and then interpreting the modal proofs as distributed programs. More specifically, the proof terms for the various modalities have computational interpretations as *remote procedure calls*, commands to *broadcast* computations to all nodes in the network, commands to use *portable* code, and finally, commands to invoke computational *agents* that can find their own way to safe places in the network where they can execute. We prove some simple meta-theoretic results about our logic as well as a safety theorem that demonstrates that the deductive rules act as a sound type system for a distributed programming language.

1 Introduction

One of the characteristics of distributed systems that makes developing robust software for them far more difficult than developing software for single stand-alone machines is *heterogeneity*. Different places in the system may have vastly different properties and resources. For instance, different machines may be attached to different hardware devices, have different software installed and run different services. Moreover, differing security concerns may cause different hosts to provide different interfaces to distributed programs, even when the underlying computational resources are similar.

In order to model such heterogeneous environments, programming language researchers usually turn to formalisms based on one sort of process algebra or another. Prime examples include the distributed join calculus [6] and the ambient calculus [3]. These calculi often come with rich theories of process equivalence and are useful tools for reasoning about distributed systems. However, a significant disadvantage of starting with process algebras as a foundation for distributed computing is that they immediately discard the wealth of logical principles that underlie conventional sequential programming paradigms and that form the sequential core of any distributed computation.

* This research was supported in part by NSF Career Award CCR-0238328 and DARPA award F30602-99-1-0519.

In this paper, we develop a foundation for safe distributed programming, which rather than rejecting the logical foundations of sequential (functional) programming, extends them with new principles tuned to programming in heterogeneous distributed environments. More specifically, we develop an intuitionistic, modal logic and provide an operational interpretation of the logical proofs as distributed programs. Our logic has the property that at any given place, all of the intuitionistic propositional tautologies are provable. Consequently, our correspondence between proofs and programs implies we can safely execute any (closed) functional program at any place in our distributed programming environment.

We extend these simple intuitionistic proofs with modal connectives and provide computational interpretations of the connectives as operations for remote code execution:

- Objects with type $\tau\,@\,p$ are return values with type τ. They are the results of *remote procedure calls* from the place p.
- Objects with type $\Box\,\tau$ are computations that run safely everywhere, and may be *broadcast* to all places in the network.
- Objects with type $\Cap\tau$ are logically equivalent to those objects with type $\Box\,\tau$, but are treated operationally simply as *portable* code that can run everywhere, but is not actually broadcast.
- Objects with type $\Diamond\tau$ are computational *agents* that have internalized the place p where they may execute safely to produce a value with type τ.

The central technical contributions of our work may be summarized as follows.

- We develop an intuitionistic, modal logic from first principles following the logical design techniques espoused by Martin Löf [12] and Frank Pfenning [14, 15]. Our logic is a relative of the hybrid logics, which are discussed in more detail at the end of the paper (see Section 4).
- Each connective in the logic is defined orthogonally to all others, is shown to be locally sound and complete, and supports the relevant substitution principles.
- This paper concentrates on the natural deduction formulation of the logic due to its correspondence with functional programs. However, we have also developed a sequent calculus that has cut elimination and shown that the sequent calculus is sound and complete with respect to the natural deduction system.
- We give an operational interpretation of the proofs in our logic as distributed functional programs. We prove that the logical deduction rules are sound when viewed as a type system for the programming language.

Due to space considerations, we have omitted many details from this extended abstract. Proofs of our theorems, language extensions (including an additional modality interpreted as place-relative remote procedure call, references and recursive functions), and further programming examples may be found in our extended technical report [9].

2 A Logic of Places

2.1 Preliminaries

The central purpose of our logic is to facilitate reasoning about heterogeneous distributed systems where different nodes may have different properties and may contain different resources. Hence, the primary judgment in the logic not only considers *whether* a formula is true, but also *where* it is true. More precisely, each primitive judgment has the form

$$\vdash^P \ F \ \mathsf{at} \ p$$

where F is a formula from the logic and p is a particular place in the system. These places are drawn from the set P. When P is unimportant or easy to guess from the context (i.e. most of the time) we omit them from the judgment and simply write $\vdash \ F \ \mathsf{at} \ p$. For any syntactic object X, $\mathsf{FP}(X)$ denote the set of free places that appear in X. We consider judgments $\vdash^P \ F \ \mathsf{at} \ p$ to have no meaning if $\mathsf{FP}(F) \cup \{p\} \not\subseteq P$.

To gain some intuition about this preliminary set up, consider a network of five computers (A, B, C, D, E). Each of these computers (A, B, C, etc.) may have different physical devices attached to it. For instance, E may have a printer attached to it and B may have a scanner attached. If sc ("there is a scanner here") and pt ("there is a printer here") are propositions in the logic, we might assert judgments such as $\vdash \ \mathsf{pt} \ \mathsf{at} \ E$ and $\vdash \ \mathsf{sc} \ \mathsf{at} \ B$ to describe this situation.

Hypothetical Judgments In order to engage in more interesting reasoning about distributed resources, we must define hypothetical judgments, facilities for reasoning from hypotheses and the appropriate notion of substitution. To begin with, hypothetical judgments have the form $\Delta \vdash^P \ F \ \mathsf{at} \ p$ where Δ is a list of (variable-labeled) assumptions:

$$\textit{contexts} \ \Delta ::= \cdot \mid \Delta, x : F \ \mathsf{at} \ p$$

We do not distinguish between contexts that differ only in the order of assumptions. We use hypotheses according to the following inference rule (where L stands for use of a Local hypothesis).

$$\frac{}{\Delta, x : F \ \mathsf{at} \ p \vdash \ F \ \mathsf{at} \ p} \ L$$

Intuitionistic Connectives With the definition of hypothetical judgments in hand, we may proceed to give the meaning of the usual intuitionistic connectives for truth ($\top$), implication ($F_1 \to F_2$) and conjunction ($F_1 \wedge F_2$) in terms of their introduction and elimination rules.

$$\frac{}{\Delta \vdash \ \top \ \mathsf{at} \ p} \ \top I$$

$$\frac{\Delta, x : F_1 \ \mathsf{at} \ p \vdash \ F_2 \ \mathsf{at} \ p}{\Delta \vdash \ F_1 \to F_2 \ \mathsf{at} \ p} \to I \qquad \frac{\Delta \vdash \ F_1 \to F_2 \ \mathsf{at} \ p \quad \Delta \vdash \ F_1 \ \mathsf{at} \ p}{\Delta \vdash \ F_2 \ \mathsf{at} \ p} \to E$$

$$\frac{\Delta \vdash F_1 \text{ at } p \quad \Delta \vdash F_2 \text{ at } p}{\Delta \vdash F_1 \wedge F_2 \text{ at } p} \wedge I$$

$$\frac{\Delta \vdash F_1 \wedge F_2 \text{ at } p}{\Delta \vdash F_1 \text{ at } p} \wedge E1 \qquad \frac{\Delta \vdash F_1 \wedge F_2 \text{ at } p}{\Delta \vdash F_2 \text{ at } p} \wedge E2$$

None of the rules above are at all surprising: Each rule helps explain how one of the usual intuitionistic connectives operates at a particular place. Hence, if we limit the set of places to a single place "_" the logic will reduce to ordinary intuitionistic logic. So far, there is no interesting way to use assumptions at multiple different places, but we will see how to do that in a moment.

As a simple example, consider reasoning about the action of the printer at place E in the network we introduced earlier. Let pdf be a proposition indicating the presence of a PDF file waiting to be printed and po be a proposition indicating the presence of a printout. The following derivation demonstrates how we might deduce the presence of a printout at E. The context Δ referenced below is

$$\mathsf{f}_E : \mathsf{pdf} \text{ at } E, \mathsf{ptr}_E : \mathsf{pt} \text{ at } E, \mathsf{print} : \mathsf{pdf} \wedge \mathsf{pt} \rightarrow \mathsf{po} \text{ at } E$$

This context represents the presence of a PDF file and printer at E as well as some software (a function) installed at E that can initiate the printing process.

$$\dfrac{\dfrac{}{\Delta \vdash \mathsf{pdf} \wedge \mathsf{pt} \rightarrow \mathsf{po} \text{ at } E}\, L \quad \dfrac{\dfrac{}{\Delta \vdash \mathsf{pdf} \text{ at } E}\, L \quad \dfrac{}{\Delta \vdash \mathsf{pt} \text{ at } E}\, L}{\Delta \vdash \mathsf{pdf} \wedge \mathsf{pt} \text{ at } E} \wedge I}{\Delta \vdash \mathsf{po} \text{ at } E} \rightarrow E$$

2.2 Inter-place Reasoning

To reason about relationships between objects located at different places we introduce a modal connective, which describes objects in terms of their location in the system.

We derive our modal connective by internalizing the judgmental notion that a formula is true at a particular place, but not necessarily elsewhere. We write this new modal formula as $F \mathbin{@} p$. The introduction and elimination rules follow.

$$\frac{\Delta \vdash F \text{ at } p}{\Delta \vdash F \mathbin{@} p \text{ at } p'} \; @\, I \qquad \frac{\Delta \vdash F \mathbin{@} p \text{ at } p'}{\Delta \vdash F \text{ at } p} \; @\, E$$

This connective allows us to reason about objects, software or devices "from a distance." For instance, in our printer example, it is possible to refer to the printer located at E while reasoning at D; to do so we might assert $\Delta \vdash \mathsf{pt} \mathbin{@} E \text{ at } D$. Moreover, we can relate objects at one place to objects at another. For instance, in order to share E's printer, D needs to have software that can convert local PDF files at D to files that may be used and print properly at E (perhaps this software internalizes some local fonts, inaccessible to E). An assumption of the form $\mathsf{DtoE} : \mathsf{pdf} \rightarrow \mathsf{pdf} \mathbin{@} E \text{ at } D$ would allow us to reason about

such software.[1] If Δ' is the assumption DtoE above together with an assumption $\mathsf{f_D} : \mathsf{pdf\ at}\ D$, the following derivation allows us to conclude that we can get the PDF file to E. We can easily compose this proof with the earlier one to demonstrate that PDF files at D can not only be sent to E, but actually printed there.

$$\dfrac{\dfrac{\overline{\Delta' \vdash \mathsf{pdf} \to \mathsf{pdf} \mathbin{@} E\ \mathsf{at}\ D}\ L \quad \overline{\Delta' \vdash \mathsf{pdf\ at}\ D}\ L}{\Delta' \vdash \mathsf{pdf} \mathbin{@} E\ \mathsf{at}\ D}\ {\to} E}{\Delta' \vdash \mathsf{pdf\ at}\ E}\ {@} E$$

2.3 Global Reasoning

While our focus is on reasoning about networks with heterogeneous resources, we cannot avoid the fact that certain propositions are true *everywhere*. For instance, the basic laws of arithmetic do not change from one machine to the next, and consequently, we should not restrict the application of these laws to any particular place. Just as importantly, we might want to reason about distributed applications deployed over a network of machines, all of which support a common operating system interface. The functionality provided by the operating system is available everywhere, just like the basic laws of arithmetic, and use of the functionality need not be constrained to one particular place or another.

Global Judgments To support global reasoning, we generalize the judgment considered so far to include a second context that contains assumptions that are valid everywhere. Our extended judgments have the form $\Gamma; \Delta \vdash^P F\ \mathsf{at}\ p$ where Γ is a global context and Δ is the local context we considered previously.

$$\begin{array}{ll} \textit{Global Contexts}\ \Gamma & ::= \cdot \mid \Gamma, x : F \\ \textit{Local Contexts} & \Delta ::= \cdot \mid \Delta, x : F\ \mathsf{at}\ p \end{array}$$

Our extended logic contains two sorts of hypothesis rules, one for using each sort of assumption. Rule L is identical to the hypothesis rule used in previous sections (modulo the unused global context Γ). Rule G specifies how to use global hypotheses; they may be placed in any location and used there.

$$\overline{\Gamma; \Delta, x : F\ \mathsf{at}\ p \vdash F\ \mathsf{at}\ p}\ L \qquad \overline{\Gamma, x : F; \Delta \vdash F\ \mathsf{at}\ p}\ G$$

All rules from the previous sections are included in the new system unchanged aside from the fact that Γ is passed unused from conclusion to premises.

Internalizing Global Truth The modal connective $\Box F$ internalizes the notion that the formula F is true everywhere. If a formula may be proven true at a new place p, which by definition can contain no local assumptions, then that formula must be true everywhere:[2]

[1] We assume that "@" binds tighter than implication or conjunction. When fully parenthesized, the assumption above has the following form: $(\mathsf{pdf} \to (\mathsf{pdf} \mathbin{@} E))\ \mathsf{at}\ D$

[2] By convention, judgments are meaningless when they contain places not contained in the sets P that annotate the judgment. Consequently, this rule requires that $p \notin \mathsf{FP}(\Gamma) \cup \mathsf{FP}(\Delta) \cup \mathsf{FP}(F)$.

$$\frac{\Gamma; \Delta \vdash^{P+p} F \text{ at } p}{\Gamma; \Delta \vdash^{P} \Box F \text{ at } p'} \Box I$$

Here, we use $P + p$ to denote the disjoint union $P \cup \{p\}$. If $p \in P$, we consider $P + p$, and any judgment containing such notation, to be undefined. If we can prove $\Box F$, we can assume that F is globally true in the proof of any other judgment F' at p':

$$\frac{\Gamma; \Delta \vdash^{P} \Box F \text{ at } p \quad \Gamma, x : F; \Delta \vdash^{P} F' \text{ at } p'}{\Gamma; \Delta \vdash^{P} F' \text{ at } p'} \Box E$$

Returning to our printing example, suppose node E decides to allow *all* machines to send it PDF files. In order to avoid hiccups in the printing process, E intends to distribute software to all machines that allow them to convert local PDF files to files that will print properly on E's printer. We might represent this situation with a hypothesis $\mathsf{ToE} : \Box\,(\mathsf{pdf} \to \mathsf{pdf} \,@\, E)$ at E. Now, given a PDF file at any other node q in the network (coded as the assumption $\mathsf{f}_q : \mathsf{pdf}$ at q), we can demonstrate that it is possible to send the PDF file to E using the following proof, where Δ'' contains assumptions ToE and f_q. The global context Γ contains the single assumption $\mathsf{ToE}' : \mathsf{pdf} \to \mathsf{pdf} \,@\, E$.

$$\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta'' \vdash \mathsf{pdf} \to \mathsf{pdf} \,@\, E \text{ at } q}\, G \quad \dfrac{}{\Gamma; \Delta'' \vdash \mathsf{pdf} \text{ at } q}\, L}{\Gamma; \Delta'' \vdash \mathsf{pdf} \,@\, E \text{ at } q} \to E}{\Gamma; \Delta'' \vdash \mathsf{pdf} \text{ at } E} @\, E}{\cdot; \Delta'' \vdash \mathsf{pdf} \text{ at } E} \Box E$$

where the derivation $\mathcal{D} = \dfrac{}{\cdot; \Delta'' \vdash \Box\,(\mathsf{pdf} \to \mathsf{pdf} \,@\, E) \text{ at } E}\, L$

The Truth is Out There The dual notion of a globally true proposition F is a proposition that is true *somewhere*, although we may not necessarily know where. We already have all the judgmental apparatus to handle this new idea; we need only internalize it in a connective ($\Diamond F$). The introduction rule states that if the formula holds at any particular place p in the network, then it holds somewhere. The elimination rule explains how we use a formula F that holds somewhere: We introduce a new place p and assume F holds there.

$$\frac{\Gamma; \Delta \vdash^{P} F \text{ at } p}{\Gamma; \Delta \vdash^{P} \Diamond F \text{ at } p'} \Diamond I \qquad \frac{\Gamma; \Delta \vdash^{P} \Diamond F \text{ at } p' \quad \Gamma; \Delta, x : F \text{ at } p \vdash^{P+p} F' \text{ at } p''}{\Gamma; \Delta \vdash^{P} F' \text{ at } p''} \Diamond E$$

Modal Axioms Possibility and necessity satisfy the following standard modal axioms (taken from Huth and Ryan [8, p. 284]).

K: $\cdot; \cdot \vdash \Box\,(F_1 \to F_2) \to (\Box\, F_1 \to \Box\, F_2)$ at p
B: $\cdot; \cdot \vdash F \to \Box \Diamond F$ at p
D: $\cdot; \cdot \vdash \Box\, F \to \Diamond F$ at p
T: $\cdot; \cdot \vdash \Box\, F \to F$ at p
4: $\cdot; \cdot \vdash \Box\, F \to \Box \Box\, F$ at p
5: $\cdot; \cdot \vdash \Diamond F \to \Box \Diamond F$ at p

2.4 Properties

We have proved local soundness and completeness properties (i.e., subject reduction for beta-reduction and eta-expansion of proofs) for each of the connectives in our logic, as well as standard substitution properties. In addition, we have developed a sequent calculus and proved the following two theorem:

Theorem 1. *The natural deduction system is sound and complete with regard to the sequent calculus (SEQ).*

Theorem 2. *The cut rule is admissible in SEQ.*

3 λ_{rpc}: A Distributed Programming Language

The previous section developed an intuitionistic, modal logic capable of concisely expressing facts about the placement of various objects in a network. Here, we present the proof terms of the logic and show how they may be given an operational interpretation as a distributed programming language, which we call λ_{rpc}. The logical formulas serve as types that prevent distributed programs from "going wrong" by attempting to access resources that are unavailable at the place where the program is currently operating.

$$
\begin{array}{l}
\textit{Types } \tau ::= \\
\mathbf{b} \mid \top \mid \tau_1 \to \tau_2 \mid \tau_1 \wedge \tau_2 \mid \tau \mathbin{@} p \mid \Box\, \tau \mid \natural \tau \mid \Diamond \tau \\
\\
\textit{Proof Terms/Programs } e ::= \\
\begin{array}{ll}
c \mid x \mid \mathbf{sync}\,(x) \mid \mathbf{run}\,(\,x\,[\,p\,]\,) \mid () & \textit{const/var/}\top \\
\mid \lambda x{:}\tau.e \mid e_1\ e_2 & \textit{functions}(\to) \\
\mid \langle\, e_1\,,\, e_2\,\rangle \mid \pi_i e & \textit{pairs}\ (\wedge) \\
\mid \mathbf{ret}(e,p) \mid \mathbf{rpc}(e,p) & \textit{rpc}\ (\,@\,) \\
\mid \mathbf{close}(\lambda p.\, e) \mid \mathbf{bc}\, e_1\ \mathsf{at}\ p\ \mathbf{as}\, x\, \mathbf{in}\, e_2 & \textit{broadcast}\ (\Box\,) \\
\mid \mathbf{port}(\lambda p.\, e) \mid \mathbf{pull}\, e_1\ \mathsf{at}\ p\ \mathbf{as}\, x\, \mathbf{in}\, e_2 & \textit{portable}\ (\natural) \\
\mid \mathbf{agent}[e,p] \mid \mathbf{go}\, e_1\ \mathsf{at}\ p\ \mathbf{return}\, x, q\, \mathbf{in}\, e_2 & \textit{agent}\ (\Diamond)
\end{array}
\end{array}
$$

Fig. 1. λ_{rpc} Syntax

3.1 Syntax and Typing

Figure 1 presents the syntax of programs and their types, and Figure 2 presents the typing rules for the language, which are the natural deduction-style proof rules for the logic.

Types and Typing Judgments. The types correspond to the formulas of the logic; we use the meta variable τ rather than F to indicate a shift in interpretation. We also included a set of base types ($\mathbf{b}$).

Since we have discovered two different operational interpretations of $\Box\, F$, and we would like to explain both of them in this paper, we have extended the language of formulas (types) to include an extra modality $\maltese\tau$ to handle the second interpretation. To support the two universal modalities, we also separate the logical context Γ into two parts, $\Gamma_{\Box}$ and $\Gamma_{\maltese}$, during type checking. Hence the overall type checking judgment has the following form.

$$\Gamma_{\Box}\,;\Gamma_{\maltese};\Delta \vdash^{P}\ e : \tau\ \mathsf{at}\ p$$

By deleting either $\Box\,\tau$ or $\maltese\tau$ (and the associated context), we recover *exactly the same logic* as discussed in the previous section.[3]

Programs. The programs include an unspecified set of constants (c), and the standard introduction and elimination forms for unit, functions and pairs.

Variables from each different context are used in different ways. As a mnemonic for the different sorts of uses, we have added some syntactic sugar to the standard proof terms. Uses of local variables from Δ are just like ordinary uses of variables in your favorite (call-by-value) functional language so they are left undecorated. Variables in $\Gamma_{\Box}$ refer to computations that have been broadcast at some earlier point. In order to use such a variable, the program must *synchronize* with the concurrently executing computation. Hence, we write $\mathbf{sync}\,(x)$ for such uses. Variables in $\Gamma_{\maltese}$ refer to portable closures. The use of a variable in this context corresponds to *running* the closure with the current place p as an argument. Hence, we write $\mathbf{run}\,(\,x\,[\,p\,]\,)$ for such uses.

Our first modality $\tau\,@\,p$ has an operational interpretation as a remote procedure call. The introduction form ($\mathbf{ret}(e, p)$) constructs a "return value" for a remote procedure call. This "return value" can actually be an arbitrary expression e, which will be returned to and run at the place p. The elimination form ($\mathbf{rpc}(e, p')$) is the remote procedure call itself. It sends the expression e to the remote site p' where e will be evaluated. If the expression is well-typed, it will eventually evaluate to $\mathbf{ret}(e', p)$: a return value that can be run safely at the caller's place, which, in this case is place p.

The introduction form for $\Box\, F$ is $\mathbf{close}(\lambda p.\, e)$. It creates a closure that may be *broadcast* by the elimination form ($\mathbf{bc}\, e_1\ \mathsf{at}\ p_1\ \mathbf{as}\, x\, \mathbf{in}\, e_2$) to every node in the network. More specifically, the elimination form executes e_1 at p_1, expecting e_1 to evaluate to $\mathbf{close}(\lambda p.\, e)$. When it does, the broadcast expression chooses a new universal reference for the closure, which is bound to x, and sends $\lambda p.e$ to every place in the network where it is applied to the current place and the resulting expression is associated with its universal reference. Finally expression e_2 is executed with the universal reference bound to x. Remote procedure calls or

[3] We could have defined two languages, one language for each interpretation of $\Box\, F$, where the typing rules of each language correspond exactly to the logical inference rules. However, we obtain a more powerful result by showing that these two interpretations inter-operate.

$$\frac{}{\Gamma_\Box; \Gamma_\boxplus; \Delta, x : \tau \text{ at } p \vdash^P x : \tau \text{ at } p}\ L \qquad \frac{}{\Gamma_\Box, x : \tau; \Gamma_\boxplus; \Delta \vdash^P \mathbf{sync}\,(x) : \tau \text{ at } p}\ G\Box$$

$$\frac{}{\Gamma_\Box; \Gamma_\boxplus, x : \tau; \Delta \vdash^P \mathbf{run}\,(\,x\,[p]\,) : \tau \text{ at } p}\ G\boxplus$$

$$\frac{}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P () : \top \text{ at } p}\ \mathit{Unit} \qquad \frac{}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P c : \mathbf{b} \text{ at } p}\ \mathit{Const}$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta, x : \tau_1 \text{ at } p \vdash^P e : \tau_2 \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \text{ at } p} \to I$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1 : \tau_1 \to \tau_2 \text{ at } p \quad \Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_2 : \tau_1 \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1\, e_2 : \tau_2 \text{ at } p} \to E$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1 : \tau_1 \text{ at } p \quad \Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_2 : \tau_2 \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \langle\, e_1\,,\, e_2\,\rangle : \tau_1 \times \tau_2 \text{ at } p} \wedge I \qquad \frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e : \tau_1 \times \tau_2 \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \pi_i e : \tau_i \text{ at } p} \wedge E$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e : \tau \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{ret}(e, p) : \tau \mathbin{@} p \text{ at } p'}\ @\, I \qquad \frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e : \tau \mathbin{@} p \text{ at } p'}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{rpc}(e, p') : \tau \text{ at } p}\ @\, E$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^{P+p} e : \tau \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{close}(\lambda p.\, e) : \Box\, \tau \text{ at } p'}\ \Box\, I$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1 : \Box\, \tau \text{ at } p \quad \Gamma_\Box, x : \tau; \Gamma_\boxplus; \Delta \vdash^P e_2 : \tau' \text{ at } p'}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{bc}\, e_1 \text{ at } p\, \mathbf{as}\, x\, \mathbf{in}\, e_2 : \tau' \text{ at } p'}\ \Box\, E$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^{P+p} e : \tau \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{port}(\lambda p.\, e) : \boxplus \tau \text{ at } p'}\ \boxplus\, I$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1 : \boxplus \tau \text{ at } p \quad \Gamma_\Box; \Gamma_\boxplus, x : \tau; \Delta \vdash^P e_2 : \tau' \text{ at } p'}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{pull}\, e_1 \text{ at } p\, \mathbf{as}\, x\, \mathbf{in}\, e_2 : \tau' \text{ at } p'}\ \boxplus\, E$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e : \tau \text{ at } p}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{agent}[e, p] : \Diamond \tau \text{ at } p'}\ \Diamond\, I$$

$$\frac{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P e_1 : \Diamond \tau \text{ at } p' \quad \Gamma_\Box; \Gamma_\boxplus; \Delta, x : \tau \text{ at } p \vdash^{P+p} e_2 : \tau' \text{ at } p''}{\Gamma_\Box; \Gamma_\boxplus; \Delta \vdash^P \mathbf{go}\, e_1 \text{ at } p'\, \mathbf{return}\, x, p\, \mathbf{in}\, e_2 : \tau' \text{ at } p''}\ \Diamond\, E$$

Fig. 2. λ_{rpc} Typing

broadcasts generated during evaluation of e_2 may refer to the universal reference bound to x, which is safe, since x has been broadcast everywhere.

Objects of type $\natural\tau$ are portable closures; they may be run anywhere. The elimination form ($\mathbf{pull}\, e_1\ \mathsf{at}\ p_1\ \mathbf{as}\, x\, \mathbf{in}\, e_2$) takes advantage of this portability by first computing e_1 at p_1, which should result in a value with the form $\mathbf{port}(\lambda p.\, e)$. Next, it pulls the closure $\lambda p.e$ from p_1 and substitutes it for x in e_2. The typing rules will allow x to appear anywhere, including in closures in e_2 that will eventually be broadcast or remotely executed. Once again, this is safe since e is portable and runs equally well everywhere.

Our last connective $\Diamond\tau$ is considered the type of a computational agent that is smart enough to know where it can go to produce a value with type τ. We introduce such an agent by packaging an expression with a place where the expression may successfully be run to completion. The elimination form ($\mathbf{go}\, e_1\ \mathsf{at}\ p_1\ \mathbf{return}\, x, p\, \mathbf{in}\, e_2$) first evaluates e_1 at p_1, producing an agent ($\mathbf{agent}[e, p_2]$). Next, it commands the agent to go to the hidden place p_2 and execute its encapsulated computation there. When the agent has completed its task, it synchronizes with the current computation and e_2 continues with p bound to p_2 and x bound to a value that is safe to use at p_2.

Simple examples. To gain a little more intuition about how to write programs in this language, consider computational interpretations of some of the proofs from the previous section. The context Δ referenced below contains the following assumptions.

$$\begin{array}{ll} \mathsf{f}_D : \mathsf{pdf}\ \mathsf{at}\ D & \mathsf{print} : \mathsf{pdf} \wedge \mathsf{pt} \to \mathsf{po}\ \mathsf{at}\ E \\ \mathsf{f}_E : \mathsf{pdf}\ \mathsf{at}\ E & \mathsf{DtoE} : \mathsf{pdf} \to \mathsf{pdf} \mathbin{@} E\ \mathsf{at}\ D \\ \mathsf{ptr}_E : \mathsf{pt}\ \mathsf{at}\ E & \mathsf{ToE} : \Box\,(\mathsf{pdf} \to \mathsf{pdf} \mathbin{@} E)\ \mathsf{at}\ E \end{array}$$

Printing a PDF file (involving local computation only):

$$\cdot; \Delta \vdash \mathsf{print}(\mathsf{f}_E, \mathsf{ptr}_E) : \mathsf{po}\ \mathsf{at}\ E$$

Fetching a PDF file (involving a remote procedure call in which the computation $\mathsf{DtoE}\ \mathsf{f}_D$ is executed at D):

$$\cdot; \Delta \vdash \mathbf{rpc}(\mathsf{DtoE}\ \mathsf{f}_D, D) : \mathsf{pdf}\ \mathsf{at}\ E$$

Fetching then printing:

$$\cdot; \Delta \vdash (\lambda x{:}\mathsf{pdf}.\mathsf{print}\ (x, \mathsf{ptr}_E))(\mathbf{rpc}(\mathsf{DtoE}\ \mathsf{f}_D, D)) : \mathsf{po}\ \mathsf{at}\ E$$

Broadcasting E's PDF conversion function to all nodes then fetching a PDF file from node q (recall that in general, uses of these global variables involves synchronizing with the broadcast expression; below the broadcast expression is a value, but we synchronize anyway):

$$\cdot; \Delta, \mathsf{f}_q : \mathsf{pdf}\ \mathsf{at}\ q \vdash \mathbf{bc}\, \mathsf{ToE}\ \mathsf{at}\ E\ \mathbf{as}\, \mathsf{ToE}'\, \mathbf{in}\, \mathbf{rpc}(\mathbf{sync}\, (\mathsf{ToE}')\ \mathsf{f}_q, q) : \mathsf{pdf}\ \mathsf{at}\ E$$

Another way to manage PDF files is to make them portable. For instance, if C and D contain portable PDF files, then E can pull these files from their resident locations and print them on its local printer. Remember that portable values are

polymorphic closures that are "run" when used. In this case, the closure simply returns the appropriate PDF file.

$$
\begin{array}{l}
\cdot; \Delta, \mathsf{f}_C : \natural\mathsf{pdf}\ \mathsf{at}\ C, \mathsf{f}_D : \natural\mathsf{pdf}\ \mathsf{at}\ D \vdash \\
\quad \mathbf{pull}\, \mathsf{f}_C\ \mathsf{at}\ C\ \mathbf{as}\, \mathsf{f}'_C\ \mathbf{in} \\
\quad \mathbf{pull}\, \mathsf{f}_D\ \mathsf{at}\ D\ \mathbf{as}\, \mathsf{f}'_D\ \mathbf{in} \\
\quad \mathbf{let}\ \_ = \mathsf{print}(\mathbf{run}\,(\,\mathsf{f}'_C\ [\,E\,]\,), \mathsf{ptr}_E)\ \mathbf{in} \\
\quad \mathbf{let}\ \_ = \mathsf{print}(\mathbf{run}\,(\,\mathsf{f}'_D\ [\,E\,]\,), \mathsf{ptr}_E)\ \mathbf{in} \\
\quad \ldots \qquad\qquad : \tau\ \mathsf{at}\ E
\end{array}
$$

3.2 Operational Semantics and Safety

When one defines an operational semantics for a language, it is essential to choose the correct level of abstraction. When the level of abstraction is too low, details get in the way of understanding the larger issues; when the level of abstraction is too high, there may be confusion as to how the semantics relates to reality.

In our case, we could give an operational semantics based directly on proof reduction, but this semantics would be at too high a level of abstraction to observe important details concerning the distribution of values over a network. In particular, we would not be able to distinguish between the two very different interpretations of $\Box$. Consequently, we give an operational semantics at a lower level of abstraction than proof reduction by including an explicit, concrete network in the semantics as shown in Figure 3.2.[4] Nevertheless, the basis for the semantics is the interaction of introduction and elimination rules as the proof theory suggests. The various new syntactic objects we use to specify our operational model are listed below.

$$
\begin{array}{lll}
\textit{Networks} & \mathcal{N} & ::= (P, \mathcal{L}) \\
\textit{Process Envs.} & \mathcal{L} & ::= \cdot \mid \mathcal{L}, \ell \to e\ \mathsf{at}\ p \\
\textit{Values} & v & ::= c \mid \lambda x{:}\tau.e \mid \langle\, v_1\,,\, v_2\,\rangle \mid \mathbf{ret}(e,p) \\
 & & \quad \mid \mathbf{close}(\lambda p.\, e) \mid \mathbf{port}(\lambda p.\, e) \mid \mathbf{agent}[e,p] \\
\textit{RT Terms} & e & ::= \cdots \mid \mathbf{sync}\,(\ell) \mid \mathbf{run}\,(\,\lambda p.e\ [\,p_1\,]\,) \mid \mathbf{sync}\,(\mathbf{rpc}(\ell,p)) \\
 & & \quad \mid \mathbf{sync}\,(\mathbf{bc}\,\ell\ \mathsf{at}\ p\ \mathbf{as}\, x\ \mathbf{in}\, e_2) \mid \mathbf{sync}\,(\mathbf{pull}\,\ell\ \mathsf{at}\ p\ \mathbf{as}\, x\ \mathbf{in}\, e_2) \\
 & & \quad \mid \mathbf{sync}_1(\mathbf{go}\,\ell\ \mathsf{at}\ p\ \mathbf{return}\, x, q\ \mathbf{in}\, e) \mid \mathbf{sync}_2(\mathbf{go}\,\ell\ \mathsf{at}\ p\ \mathbf{return}\, x, q\ \mathbf{in}\, e) \\
\textit{Contexts} & C & ::= [\,] \mid C\, e_2 \mid v_1\, C \mid \langle\, C\,,\, e_2\,\rangle \mid \langle\, v_1\,,\, C\,\rangle \mid \pi_i\, C
\end{array}
$$

Networks $\mathcal{N}$ are pairs consisting of a set of places P, a distributed process environment $\mathcal{L}$. We have seen places before. The process environment $\mathcal{L}$ is a

[4] Our choice here is also informed by the history of operational interpretations of linear logic. Several researchers [1, 11] gave interpretations of linear logic using a store-less semantics derived directly from logical proof reduction. These semantics led to *significant* confusion about the memory management properties of the languages: How many pointers to a linear object could there be, and can linear objects be deallocated after they are used? It was impossible to say because the operational models did not contain pointers! Only when Chirimar et al. [4] and Turner and Wadler [18] later gave interpretations that deviated from the proof-theoretic interpretation by including an explicit store was the story made clear. Our interpretation of modal logic deviates from the proof theory in a similar way to these latter works as we include an explicit network.

$\mathcal{L} \longmapsto \mathcal{L}'$	
sync *OS*	$\mathcal{L}, \ell' \to C\,[\,\mathbf{sync}\,(\ell)\,]$ at p, $\ell \to v$ at p $\longmapsto \mathcal{L}, \ell' \to C\,[\,v\,]$ at p, $\ell \to v$ at p
run OS	$\mathcal{L}, \ell \to C\,[\,\mathbf{run}\,(\,\lambda p.e\,[\,p_1\,]\,)\,]$ at p_2 $\longmapsto \mathcal{L}, \ell \to C\,[\,e\,[\,p_1\,/\,p\,]\,]$ at p_2
$\to$ *OS*	$\mathcal{L}, \ell \to C\,[\,(\lambda x{:}\tau.e)\,v\,]$ at p $\longmapsto \mathcal{L}, \ell \to C\,[\,e\,[\,v\,/\,x\,]\,]$ at p
$\wedge$ *OS*	$\mathcal{L}, \ell \to C\,[\,\pi_i\langle\, v_1\,,\, v_2\,\rangle\,]$ at p $\longmapsto \mathcal{L}, \ell \to C\,[\,v_i\,]$ at p
@ *OS1*	$\mathcal{L}, \ell \to C\,[\,\mathbf{rpc}(e, p_1)\,]$ at p_0 $\longmapsto \mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{rpc}(\ell_1, p_1))\,]$ at p_0, $\ell_1 \to e$ at p_1
@ *OS2*	$\mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{rpc}(\ell_1, p_1))\,]$ at p_0, $\ell_1 \to \mathbf{ret}(e, p_0)$ at p_1 $\longmapsto \mathcal{L}, \ell \to C\,[\,e\,]$ at p_0, $\ell_1 \to \mathbf{ret}(e, p_0)$ at p_1
$\Box$ *OS1*	$\mathcal{L}, \ell \to C\,[\,\mathbf{bc}\,e_1$ at p_1 **as** x **in** $e_2\,]$ at p_0 $\longmapsto \mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{bc}\,\ell_1$ at p_1 **as** x **in** $e_2)\,]$ at p_0, $\ell_1 \to e_1$ at p_1
$\Box$ *OS2*	$\mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{bc}\,\ell_1$ at p_1 **as** x **in** $e_2)\,]$ at p_0, $\ell_1 \to \mathbf{close}(\lambda p.\,e)$ at p_1 $\longmapsto \mathcal{L}, \ell \to C\,[\,e_2\,[\,\ell_2\,/\,x\,]\,]$ at p_0, $\ell_1 \to \mathbf{close}(\lambda p.\,e)$ at p_1, $\{\,\ell_2 \to e\,[\,q\,/\,p\,]$ at $q\,\}$ $(\forall q \in P)$
$\boxtimes$ *OS1*	$\mathcal{L}, \ell \to C\,[\,\mathbf{pull}\,e_1$ at p_1 **as** x **in** $e_2\,]$ at p_0 $\longmapsto \mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{pull}\,\ell_1$ at p_1 **as** x **in** $e_2)\,]$ at p_0, $\ell_1 \to e_1$ at p_1
$\boxtimes$ *OS2*	$\mathcal{L}, \ell \to C\,[\,\mathbf{sync}\,(\mathbf{pull}\,\ell_1$ at p_1 **as** x **in** $e_2)\,]$ at p_0, $\ell_1 \to \mathbf{port}(\lambda p.\,e)$ at p_1 $\longmapsto \mathcal{L}, \ell \to C\,[\,e_2\,[\,\lambda p.e\,/\,x\,]\,]$ at p_0, $\ell_1 \to \mathbf{port}(\lambda p.\,e)$ at p_1
$\diamond$ *OS1*	$\mathcal{L}, \ell \to C\,[\,\mathbf{go}\,e_1$ at p_1 **return** x, q **in** $e_2\,]$ at p_0 $\longmapsto \mathcal{L}, \ell \to C\,[\,\mathbf{sync}_1(\mathbf{go}\,\ell_1$ at p_1 **return** x, q **in** $e_2)\,]$ at p_0, $\ell_1 \to e_1$ at p_1
$\diamond$ *OS2*	$\mathcal{L}, \ell \to C\,[\,\mathbf{sync}_1(\mathbf{go}\,\ell_1$ at p_1 **return** x, q **in** $e_2)\,]$ at p_0, $\ell_1 \to \mathbf{agent}[e, p_2]$ at p_1 $\longmapsto \mathcal{L}, \ell \to C\,[\,\mathbf{sync}_2(\mathbf{go}\,\ell_2$ at p_2 **return** x, q **in** $e_2)\,]$ at p_0, $\ell_1 \to \mathbf{agent}[e, p_2]$ at p_1, $\ell_2 \to e$ at p_2
$\diamond$ *OS3*	$\mathcal{L}, \ell \to C\,[\,\mathbf{sync}_2(\mathbf{go}\,\ell_1$ at p_1 **return** x, q **in** $e_2)\,]$ at p_0, $\ell_1 \to v$ at p_1 $\longmapsto \ell \to C\,[\,e_2\,[\,p_1\,/\,q\,]\,[\,v\,/\,x\,]\,]$ at p_0, $\ell_1 \to v$ at p_1

Fig. 3. λ_{rpc} Operational Semantics

finite partial map from places p in P to process ids to expressions. We write these partial maps as lists of elements with the form $\ell \to e$ at p. We assume that no pair of place and location (p and ℓ) appears in two different components of the map. We do not distinguish between maps that differ only in the ordering of their elements. $\mathcal{L}(p)(\ell)$ denotes e when $\mathcal{L} = \mathcal{L}', \ell \to e$ at p. We use the notation $\mathcal{L} \setminus \ell$ to denote the mapping $\mathcal{L}$ with all elements of the form $\ell \to e$ at p removed.

We also introduce new expressions (the RT terms above) that only occur at run time to give an operation semantics to our program. Run-time terms are used to represent expressions, which are suspended part-way through evaluation and are waiting to synchronize with remotely executing expressions.

Lastly, we define evaluation contexts C to specify the order of evaluation.

In order to show that the network is well-typed at every step in evaluation, we add typing rules to give types to the RT terms and we also give well-formedness

conditions for the network as a whole. The typing judgment for a network has the form $\vdash \mathcal{L} : \Gamma_\Box; \cdot; \Delta$. See the technical report [9] for further details.

Operational Rules and Type Safety. The state of a network $\mathcal{N} = (P, \mathcal{L})$ evolves according to the operational rules listed in Figure 3.2. These rules specify a relation with the form $\mathcal{L} \longmapsto \mathcal{L}'$.

The type system is sound with respect to our operational semantics for distributed program evaluation. The proofs of Preservation and Progress theorems, stated below, follow the usual strategy.

Theorem 3 (Preservation). *If* $\vdash \mathcal{L} : \Gamma_\Box; \cdot; \Delta$ *and* $\mathcal{L} \longmapsto \mathcal{L}'$ *then there exists* $\Gamma'_\Box$ *and* Δ' *such that* $\vdash \mathcal{L}' : \Gamma'_\Box; \cdot; \Delta'$.

Theorem 4 (Progress). *If* $\vdash \mathcal{L} : \Gamma_\Box; \cdot; \Delta$ *then either*

- $\mathcal{L} \longmapsto \mathcal{L}'$, *or*
- *for all places* p *in* P, *and for all* ℓ *in* $\mathsf{Dom}(\mathcal{L}(p))$, $\mathcal{L}(p)(\ell)$ *is a value.*

4 Discussion

Extensions and Variations This paper presents a solid foundation on which to build a distributed functional programming language. However, in terms of language design, it is only the beginning. A few interesting extensions and variations are discussed briefly below.

- Values everywhere. Our interpretation of $\Box$ involves either broadcasting a closure or substituting a closure into local code. In each case, there is some computational overhead to manage the closure: Either we synchronize or we run the closure when it gets used. To avoid this overhead, we could place a value restriction on the expression in the introduction form for $\Box$. One possibility that is definitely not an option is evaluating eagerly under $\Box$ before broadcasting or substitution: In the presence of references (and almost certainly other effects), this evaluation strategy is unsound.
- Dynamic network evolution. The current work assumes that the set of network places and the network topology is fixed. While this is a reasonable assumption for some distributed programming environments, others allow the topology to evolve. An interesting challenge for future work is to extend our logic and language with features that express evolution. We believe that the new name connectives developed in the context of nominal logics [16, 13] may be of help here.
- Synchronous and asynchronous variations. Just as ordinary sequential programming languages may be defined with different evaluation strategies (call-by-value, call-by-name, call-by-need), it appears possible to develop different operational interpretations of the modal connectives in which execution is more or less synchronized. For instance, when defining the operation of the $\Box$-connective, we could wait until all broadcast expressions have completed evaluation before proceeding with the evaluation of the second expression e_2. Likewise, remote procedure calls are synchronized: Evaluation does not

proceed until they have received the return value, even though the following computation does not necessarily require the value immediately. In the future, we plan to explore these nuances in greater detail.

Related Work Hybrid logics are an old breed of logic that date back to Arthur Prior's work in the 1960s [17]. As does our logic, they mix modal necessity and possibility with formulas such as $F \mathbin{@} p$ that are built from pure names. More recently, researchers have developed a rich semantic theory for these logics and studied both tableau proofs and sequents; many resources on these topics and others are available off the hybrid logics web page.[5] However, work on hybrid logics is usually carried out in a classical setting and we have not found an intuitionistic, natural deduction style proof theory like ours that can serve as a foundation for distributed functional programming languages. The reason for this is likely that Pfenning and Davies [15] have only recently shown how to assign a robust natural deduction-style proof theory to pure modal logic.

Cardelli and Gordon's ambient logic [2] highlights the idea that modalities for possibility and necessity need not only be interpreted *temporally*, but can also be interpreted *spatially*, and this abstract idea was a central influence in our work. However, at a more technical level, the ambient logic is entirely different from the logic we develop here: The ambient logic has a widely different set of connectives, is classical as opposed to intuitionistic, and is defined exclusively by a sequent calculus rather than by natural deduction. Moreover, it does not serve as a type system for ambient programs; rather, it is a tool for reasoning about them.

Much effort has been invested in interpreting proof search in sequent calculus of linear logic as concurrent computation. Recently along this line of research, Kobayshi et al. have explained how modal linear logic can be viewed as a distributed concurrent programming language [10]. In this work, they introduce a formula similar to our $F \mathbin{@} p$ but they define it axiomatically rather than through introduction and elimination rules (or left and right rules), as we have done. Consequently, their formulation cannot serve as the basis for a functional programming language. They did not consider modal necessity and possibility.

Another major influence on our work is Pfenning and Davies' judgmental reconstruction of modal logic [15], which is developed in accordance with Martin Löf's design patterns for type theory [12]. Pfenning and Davies go on to interpret modal necessity temporally (as opposed to spatially) in their work on staged computation [5]. One obvious technical difference between our logic and theirs is that our logic is founded on local judgments that include the specific place where a proposition is true whereas their logic is not.

Concurrently with this research, Harper, Moody and Pfenning [7] have begun to investigate the role that modal logics may play in developing a foundation for distributed computing. More specifically, they interpret objects with type $\Box\,\tau$ as jobs that may be injected into the Grid and run anywhere. This interpretation

[5] See `http://www.hylo.net`.

appears to resemble our portable code more closely than our broadcast code. In addition, in their Grid computing domain, every node is assumed to contain identical resources, so they are investigating type systems derived from pure modal logics rather than hybrid logics like the one we have presented here. Consequently, they have no analogue of our remote procedure calls.

References

1. S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. L. Cardelli and A. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, Jan. 2000.
3. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.
4. J. Chirimar, C. A. Gunter, and J. G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Apr. 1992.
5. R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.
6. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, Aug. 1996. Springer.
7. R. Harper. Trustless grid computing in concert, June 2003. Talk. http://www-2.cs.cmu.edu/ concert/talks/Harper2003Trustless/trustless.ppt.
8. M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.
9. L. Jia and D. Walker. Modal proofs as distributed programs. Technical Report TR-671-03, Princeton University, 2003. Forthcoming.
10. N. Kobayashi, T. Shimizu, and A. Yonezawa. Distributed concurrent linear logic programming. *Theoretical Computer Science*, 227(1–2):185–220, Sept. 1999.
11. P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Proceedings 7th Annual IEEE Symp. on Logic in Computer Science, LICS'92, Santa Cruz, CA, USA, 22–25 June 1992*, pages 235–246. IEEE Computer Society Press, Los Alamitos, CA, 1992.
12. M. Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, University of Siena, 1985.
13. D. Miller and A. Tiu. A proof theory for generic judgments. In *IEEE Symposium on Logic in Computer Science*, pages 118–127, Ottawa, Canada, June 2003.
14. F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001.
15. F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.
16. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, 2003. To appear.
17. A. Prior. *Past, present and future*. Oxford University press, 1967.
18. D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.