Modal Logic as a Basis for Distributed Computation (DRAFT)

Jonathan Moody
jwmoody+@cs.cmu.edu

February 18, 2003

1 Introduction

In this report, we give a computational interpretation of modal logic in which the modalities necessity ($\Box$) and possibility ($\Diamond$) are used to describe locality in a distributed computation. This interpretation is quite natural, given the usual "possible worlds" model underlying modal logic. In our case, the possible worlds we will consider are nodes participating in a distributed computation. The necessity modality ($\Box$) will describe a term that may be evaluated (safely) *anywhere* and possibility ($\Diamond$) a term that may be evaluated *somewhere*. In this sense, types will determine the permissible degree of mobility for terms.

Type	Interpretation
A	term of type A *here*
$\Box A$	term of type A *any* place
$\Diamond A$	term of type A *some* place

In addition to the purely logical motivations, we present some examples demonstrating how the language of modal logic proof terms allows us to write distributed, concurrent programs while preserving safe access and manipulation of localized resources. This work is supported by the NSF GRFP[1], as well as the CMU ConCert[2] project.

[1] "This material is based upon work supported under a National Science Foundation Graduate Research Fellowship."

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

2 Modal Logic

Modal logic comes in many flavors – the flavor used in this report is intuitionistic S4. For an introduction to this system of modal logic see [8] by Pfenning and Davies. In later sections of [8], the authors provide a language of proof terms, which can be interpreted as programs via the Curry-Howard isomorphism. We adopt their notation for this work also.

This language of proof terms would be somewhat awkward for programming; in particular, the names "box" and "dia" have no meaning to programmers. However, this approach has one advantage. If we were to start with more meaningful names, it would mean that in some sense, we had already decided what these operators should do. The very meaninglessness of "box" and "dia" provides a blank slate on which to brainstorm, guided only by the logical meanings of $\Box$ and $\Diamond$. One must keep in mind, however, that this process of "reading the tea leaves" can only provide a rough sketch of the solution, because there may be many type-sound evaluation strategies.

In this report, we develop a reasonable, logically-motivated operational interpretation of modal logic proof terms. The precise formulation of the operational semantics is somewhat underdetermined given only the logical properties of the language. However, by working from both the logical and the engineering ends of the problem, we show that modal logic meshes nicely with the practical requirements of a language for distributed programming. Our results represent one interpretation that we judged best under various practical constraints and desiderata.

3 Proof Language

The following term assignment for modal logic is reproduced from [8]. The development of Pfenning and Davies was based on three primitive judgements $A\,\texttt{valid}$, $A\,\texttt{true}$, and $A\,\texttt{poss}$, with meanings A is true "everywhere", "here", and "somewhere". However, only $A\,\texttt{true}$ and $A\,\texttt{poss}$ are needed to explain the typing rules for the proof language, because $A\,\texttt{valid}$ is defined as deduction of $A\,\texttt{true}$ from no assumptions.

$$
\begin{array}{rcl}
\text{Variable } X & ::= & \texttt{x} \mid \texttt{y} \mid \ldots \\
\text{Valid Variable } U & ::= & \texttt{u} \mid \texttt{v} \mid \ldots \\
\text{Term } M, N & ::= & X \mid U \mid \lambda \texttt{x} : A \,.\, M \mid M\ N \\
& & \mid \texttt{box}\, M \mid \texttt{let box u} = M\ \texttt{in}\ N \\
& & \mid \texttt{dia}\, E \\
\text{Expression } E, F & ::= & \{M\} \mid \texttt{let box u} = M\ \texttt{in}\ F \\
& & \mid \texttt{let dia x} = M\ \texttt{in}\ F
\end{array}
$$

The distinction between terms and expressions can be motivated purely logically. The expressions are simply those which are proofs of $A\,\texttt{poss}$, whereas

terms are those which prove $A\,\mathtt{true}$. The inclusion of terms in the category of expressions (as $\{M\}$) is a consequence of the logical inclusion between truth and possibility.

The form of the typing judgment for terms will be $\Delta;\Gamma \vdash M : A$, where Δ and Γ are variable typing contexts for *valid* and *true* hypotheses, respectively. Valid hypotheses in Δ are globally available (in all subterms and subexpressions), whereas the locally true hypotheses in Γ obey additional scoping restrictions. Note that the unconventional expression typing judgement $\Delta;\Gamma \vdash E \div A$ is a notation meaning "expression E proves $A\,\mathtt{poss}$".

$$\begin{array}{rrcl} \text{Types} & A, B & ::= & A \to B \mid \Box A \mid \Diamond A \\ \text{Valid Context} & \Delta & ::= & \cdot \mid \Delta, U :: A \\ \text{True Context} & \Gamma & ::= & \cdot \mid \Gamma, X : A \end{array}$$

$$\frac{}{\Delta;\Gamma,\mathtt{x} : A,\Gamma' \vdash \mathtt{x} : A}\ hyp \qquad \frac{}{\Delta,\mathtt{u} :: A,\Delta';\Gamma \vdash \mathtt{u} : A}\ hyp^*$$

$$\frac{\Delta;\Gamma,\mathtt{x} : A \vdash M : B}{\Delta;\Gamma \vdash \lambda\mathtt{x} : A\,.\,M : A \to B} \to I \qquad \frac{\Delta;\Gamma \vdash M : A \to B \quad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M\ N : B} \to E$$

$$\frac{\Delta;\cdot \vdash M : A}{\Delta;\Gamma \vdash \mathtt{box}\,M : \Box A} \Box I \qquad \frac{\Delta;\Gamma \vdash M : \Box A \quad \Delta,\mathtt{u} :: A;\Gamma \vdash N : B}{\Delta;\Gamma \vdash \mathtt{let\ box\,u} = M\,\mathtt{in}\,N : B} \Box E$$

$$\frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash \{M\} \div A}\ poss \qquad \frac{\Delta;\Gamma \vdash M : \Box A \quad \Delta,\mathtt{u} :: A;\Gamma \vdash F \div B}{\Delta;\Gamma \vdash \mathtt{let\ box\,u} = M\,\mathtt{in}\,F \div B} \Box E_p$$

$$\frac{\Delta;\Gamma \vdash E \div A}{\Delta;\Gamma \vdash \mathtt{dia}\,E : \Diamond A} \Diamond I \qquad \frac{\Delta;\Gamma \vdash M : \Diamond A \quad \Delta;\mathtt{x} : A \vdash F \div B}{\Delta;\Gamma \vdash \mathtt{let\ dia\,x} = M\,\mathtt{in}\,F \div B} \Diamond E$$

Note that in rule $\to I$, we treat the new bound variable $\mathtt{x}$ as a "locally true" hypothesis. This indicates that variables ($\mathtt{x}$) correspond to the usual notion of variables in the lambda-calculus, whereas the valid hypotheses ($\mathtt{u}$) are fundamentally more powerful. The typing rules $\Box I$ and $\Diamond E$ deserve special attention, because they impose unusual scoping restrictions on variables of the ordinary, locally true variety. In later sections we will argue that these restrictions are essential to justify mobility.

4 An Operational Semantics

In this section, we will extend the work of [8], in which various logically sound reduction rules were derived, to an operational semantics which clearly reflects locality in evaluation. We must keep in mind both the desired informal interpretations of $\Box A$ and $\Diamond A$, as well as formal judgemental definitions of these

propositions which should provide some clue as to why we are allowed to move certain proof terms between locations while others must remain fixed.

Before proceeding to analyze the behavior of terms and expressions in detail, a few words about the static properties of terms (establishing truth) and expressions (establishing possibility) are in order. One must keep in mind that expressions, being evidence for $A\,\mathtt{poss}$, are those things which have (or acquire during evaluation) a location-dependent meaning. These proofs of $A\,\mathtt{poss}$ must follow a certain strict pattern of reasoning, using only what is know to be true at one particular world, producing a conclusion at the same world. Terms are the more mundane, location-neutral objects in the language. In reasoning toward $A\,\mathtt{true}$ we are allowed to use whatever else is known in the current location. Terms are only localized to the extent that they depend on other locally true hypotheses in Γ; in fact, terms which are closed with respect to Γ have a *universal* meaning — they establish $A\,\mathtt{valid}$.

It is also important to note that we interpret the typing judgement as describing where a term *will be* well-formed, rather than where a term resides at a particular moment. It is natural to assume that all parts of a program will reside at a single location initially, but as that program evolves under evaluation, fragments will be spawned for evaluation at other locations. The key runtime property that the operational semantics should obey is that no term or expression is evaluated (or interpreted in any way) until it is placed in the proper context.

4.1 Justification of Mobility

Consider the unusual form of some of the typing rules, namely $\Box I$ and $\Diamond E$. We will argue that the restrictions they impose on the form of Γ (the ordinary variables) provides the logical justification we need to make parts of a program mobile.

$$\frac{\Delta;\cdot \vdash M : A}{\Delta;\Gamma \vdash \mathtt{box}\, M : \Box A}\ \Box I \qquad \frac{\Delta;\Gamma \vdash M : \Box A \quad \Delta, \mathtt{u} :: A;\Gamma \vdash N : B}{\Delta;\Gamma \vdash \mathtt{let\ box\,u} = M\ \mathtt{in}\ N : B}\ \Box E$$

$$\frac{\Delta;\Gamma \vdash M : \Box A \quad \Delta, \mathtt{u} :: A;\Gamma \vdash F \div B}{\Delta;\Gamma \vdash \mathtt{let\ box\,u} = M\ \mathtt{in}\ F \div B}\ \Box E_p$$

In the case of $\Box I$, reading the rule from the bottom up, if we have a term $\mathtt{box}\, M$ proving $\Box A$, we must have a closed term M of type A, which is valid *anywhere*. That is, M depends on no locally true assumptions in Γ. Therefore it makes sense to treat M as being fully mobile; we may choose to send M to any location at runtime. Under the elimination rules $\Box E$ and $\Box E_p$, we see that given $M = \mathtt{box}\, M'$ of type $\Box A$, we may rely on the valid hypothesis $\mathtt{u} :: A$ throughout the remainder of the program (N or F). Essentially, the boxed term M' becomes globally available. Hence we see that any operational semantics must provide a mechanism to give any location access to M' (or the value of M') by reference to $\mathtt{u} :: A$. This is the intuition behind the behavior of necessity ($\Box$).

$$\frac{\Delta;\Gamma \vdash E \div A}{\Delta;\Gamma \vdash \mathtt{dia}\, E : \Diamond A}\ \Diamond I \qquad \frac{\Delta;\Gamma \vdash M : \Diamond A \quad \Delta;\mathtt{x} : A \vdash F \div B}{\Delta;\Gamma \vdash \mathtt{let\ dia\, x = } M \,\mathtt{in}\, F \div B}\ \Diamond E$$

Now in the case of $\Diamond I$, reading the rule from the bottom up, forming a term $\mathtt{dia}\, E$ of type $\Diamond A$ requires that we have an expression $\Delta;\Gamma \vdash E \div A$. That is, from a perspective where we know hypotheses in Γ are true, E proves A, possibly somewhere else. The particular location is not made clear, but the important thing to note is that E is *fixed* to that location — it must be regarded as totally immobile. For the elimination form $\Diamond E$, reading from top to bottom, we will have a term $M = \mathtt{dia}\, E$ with type $\Diamond A$ and an expression F such that $\Delta;\mathtt{x} : A \vdash F \div B$. As remarked above, we have in mind some particular fixed location where E should be evaluated. Furthermore, we know $F \div B$ under the assumption $\mathtt{x} : A$. Because the judgement $\Delta;\mathtt{x} : A \vdash F \div B$ depends only on a single true hypothesis $\mathtt{x} : A$, it makes sense to claim that F is mobile in a restricted sense; that is, we may send F to the particular world where E proves A, validating the assumption $\mathtt{x} : A$. By doing so we will have established $B\,\mathtt{poss}$ as required. This is the intuition behind the behavior of possibility ($\Diamond$).

4.2 Representing Locality

We now introduce a notation for processes and process labels. Besides providing a mechanism for concurrent evaluation, processes are also identified with the "worlds" at which computation takes place. This is not a very general notion of location, for example, we cannot group processes by location or represent hierarchies of locations as in Cardelli's ambient notation [3]. Here we are limited by the language in which we express programs. Because worlds are not present in the language and typing judgement, there is no sensible (type-theoretic) way of concluding that two terms or processes should have the same location.[3]

It is also clear we will need some form of process labels, so that processes may refer to each other. However, we distinguish between strong labels (r_i) corresponding to logical validity, which we call "result labels" and weak "location labels" (l_i) corresponding roughly to logical possibility. Result labels will allow us to receive the result value of a process, whereas location labels are placeholders for "detached" processes which we can no longer access.

$$\begin{array}{rrcl}
\text{Result Label} & R & ::= & r_0 \mid r_1 \mid \ldots \\
\text{Location Label} & L & ::= & l_0 \mid l_1 \mid \ldots \\
\text{Label} & W & ::= & R \mid L \\
\text{Process} & P & ::= & \langle R : M \rangle \mid \langle L : E \rangle \mid \langle L : L' \rangle \\
\text{Configuration} & C & ::= & \cdot \mid C; P
\end{array}$$

Note that location labels l_i, though associated with logical possibility, are not formally expressions. Hence the need for a special form of process $\langle L :$

[3]There are, however, other formulations of intuitionistic modal logic which incorporate worlds explicitly into proof terms. Investigation of these formalisms is ongoing.

$L'\rangle$ representing the effect of evaluating an expression. We will assume an equivalence relation defined on process configurations $C \equiv C'$ such that $\equiv$ is a congruence satisfying associativity and commutativity over parallel composition (;). This standard technical mechanism permits a concise specification of the operational semantics.

4.3 Logical Characterization of Labels

As mentioned before, the two forms of label r_i and l_i should be interpreted logically as validity and possibility. To do this, we extend the typing judgement with a process typing context Λ containing assumptions about the type of labels W occurring in the program. We say a process configuration C has type Λ, written as $\vdash_C C : \Lambda$, if it can be derived from the following rules (modulo equivalence $C \equiv C'$).

$$\text{Label Context} \quad \Lambda \quad ::= \quad \cdot \quad | \quad \Lambda, R :: A \quad | \quad \Lambda, L \div A$$

$$\frac{\vdash_C C : \Lambda \quad \Lambda; \cdot; \cdot \vdash M : A}{\vdash_C C; \langle r_i : M \rangle : \Lambda, r_i :: A}\ pterm \qquad \frac{\vdash_C C : \Lambda \quad \Lambda; \cdot; \cdot \vdash E \div A}{\vdash_C C; \langle l_i : E \rangle : \Lambda, l_i \div A}\ pexp \qquad \frac{}{\vdash_C \cdot : \cdot}$$

Note that this definition implies that there can be no cycles in the use of process labels in well-formed process configurations. Proving $\vdash_C C : \Lambda$ requires choosing a linear order (consistent with the occurrence of labels) in which the processes will be typed.

With the foregoing definition of a label typing Λ in mind, we extend the language of terms M and term typing to allow for result labels:

$$\frac{}{\Lambda, r_i :: A, \Lambda'; \Delta; \Gamma \vdash r_i : A}\ res$$

Location labels, on the other hand, will *not* be treated as expressions, though in an informal sense $l_i \div A$. Due to a peculiarity[4] of the operational semantics, location labels will only occur as top-level placeholders for expressions. That is, location labels will appear as a special form of process $\langle l_j : l_i \rangle$ but will not themselves be expressions E. Hence location labels are accounted for in the process configuration typing judgement, $\vdash_C C : \Lambda$.

$$\frac{\vdash_C C : \Lambda \quad \Lambda = \Lambda_1, l_i \div A, \Lambda_2}{\vdash_C C; \langle l_j : l_i \rangle : \Lambda, l_j \div A}\ ploc$$

The typing rules for terms and expressions are extended in a straightforward way to carry a label typing Λ. Of course the label context Λ will not interact with term or expression typing except through the rule *res*.

[4]This property of location labels is a consequence of the particular evaluation strategy used, and is not a logical necessity. However, treating location labels as expressions complicates the operational semantics and the proof of a progress theorem.

4.4 Values and Canonical Forms

Although other interpretations can lead to increased concurrency during execution, we chose to treat $\texttt{box}\,M$ and $\texttt{dia}\,E$ as values. This leads to a less rich runtime behavior, but simplifies the semantics considerably. Two judgements, M `tvalue` and E `evalue`, define the form of term and expression values, respectively.

$$\frac{}{\lambda\texttt{x} : A\,.\,M\ \texttt{tvalue}} \qquad \frac{}{\texttt{box}\,M\ \texttt{tvalue}} \qquad \frac{}{\texttt{dia}\,E\ \texttt{tvalue}} \qquad \frac{}{r_i\ \texttt{tvalue}} \qquad \frac{V\ \texttt{tvalue}}{\{V\}\ \texttt{evalue}}$$

Note that the expression values are merely coerced term values $\{V\}$ in this formulation of the language. The canonical forms principle for value typing is as follows:

$$\begin{array}{lcll} V\ \texttt{tvalue} & \wedge & \vdash V : A \to B & \Longrightarrow \quad V = \lambda\texttt{x} : A\,.\,M \ \vee\ V = r_i \\ V\ \texttt{tvalue} & \wedge & \vdash V : \Box A & \Longrightarrow \quad V = \texttt{box}\,M \ \vee\ V = r_i \\ V\ \texttt{tvalue} & \wedge & \vdash V : \Diamond A & \Longrightarrow \quad V = \texttt{dia}\,E \ \vee\ V = r_i \\ \\ V\ \texttt{evalue} & \wedge & \vdash V \div A & \Longrightarrow \quad V = \{V'\} \ \wedge\ V'\ \texttt{tvalue} \wedge \vdash V : A \end{array}$$

Note that term values do not have a unique canonical form. In any context $V : A$ we may encounter a result label (r_i), as well as the usual canonical forms for a value of type A. Expression values $(V \div A)$ for any type A will have the form $\{V'\}$.[5]

4.5 Definition of Substitution

Pfenning and Davies develop a substitution-based notion of reduction in their paper [8]. Substitution of terms for ordinary and valid variables ($[M/\texttt{x}]N$ and $[[M/\texttt{u}]]N$ respectively), were defined as one would expect, taking into account restrictions on the scope of ordinary variables. However, they found that an unusual definition of substitution on expressions was necessary in order to maintain type soundness. Substitution of expressions into expressions (including terms) was defined as follows:

$$\begin{array}{rcl} \langle\langle\{M\}/\texttt{x}\rangle\rangle F & = & [M/\texttt{x}]F \\ \langle\langle\texttt{let dia y} = M\ \texttt{in}\ E/\texttt{x}\rangle\rangle F & = & \texttt{let dia y} = M\ \texttt{in}\ \langle\langle E/\texttt{x}\rangle\rangle F \\ \langle\langle\texttt{let box u} = M\ \texttt{in}\ E/\texttt{x}\rangle\rangle F & = & \texttt{let box u} = M\ \texttt{in}\ \langle\langle E/\texttt{x}\rangle\rangle F \end{array}$$

Note that the definition of $\langle\langle E/\texttt{x}\rangle\rangle F$ is inductive in the structure of E rather than F. This form of substitution is applied to reduce $\Diamond$ introduction/elimination. An inspection of the typing rule $\Diamond E$ shows why substitution must behave this way. Specifically, F is well-formed under the assumption $\texttt{x} : A$, that is, x is assumed to be a term. Simply replacing x with E would not, in general, result in a well-formed expression.

[5] This property will, of course, cease to hold in a language extended with additional forms of primitive expression.

We have extended the syntax of terms with result labels. Hence it is technically necessary to extend the definition of substitution as well. Result labels are dealt with in the obvious way:

$$\begin{aligned} [M/\mathtt{x}]r_i &= r_i \\ [[M/\mathtt{u}]]r_i &= r_i \end{aligned}$$

Note that we need *not* define substitution of location labels l_i, because they are not formally expressions. In particular, we avoid having to define $\langle\langle l_i/\mathtt{x}\rangle\rangle F$ which would be problematic.

4.6 Transition Rules

A single-step transition in the semantics is stated as $C \Rightarrow C'$ for process configurations C and C'. Processes irrelevant to the transition are omitted; that is, $C; \langle l_i : E\rangle; C'$ is abbreviated as $\langle l_i : E\rangle$. We will be using the notation of evaluation contexts $\mathcal{S}$ to reflect where (in a term or expression) reduction may take place. In fact, evaluation contexts can be split into two definitions, term and expression contexts.

$$\begin{aligned} \text{Term Context} \quad \mathcal{R} &::= [\,] \mid \mathcal{R}\ M \mid V\ \mathcal{R} \\ &\quad \mid \mathtt{let\ box\,u} = \mathcal{R}\ \mathtt{in}\ N \\ \text{Expression Context} \quad \mathcal{S} &::= [\,] \mid \{R\} \\ &\quad \mid \mathtt{let\ box\,u} = \mathcal{R}\ \mathtt{in}\ E \\ &\quad \mid \mathtt{let\ dia\,x} = \mathcal{R}\ \mathtt{in}\ E \end{aligned}$$

Note that only terms M may appear in a context $\mathcal{R}[\,M\,]$. Note also that the structure of $\mathcal{S}$ implies we will only perform reductions on true expressions (those which are not also terms) in the empty context ($\mathcal{S} = [\,]$).

Rules for reduction of terms will occur in pairs, one applicable to processes of the form $\langle r_i : \mathcal{R}[\,M\,]\rangle$, the other for processes $\langle l_i : \mathcal{S}[\,M\,]\rangle$. We follow a convention of naming these variants *app*, *app′*, etc.

$$\frac{V_1 = (\lambda \mathtt{x} : A\,.\,M') \quad V_2\ \mathtt{tvalue}}{\langle r_i : \mathcal{R}[\,V_1\ V_2\,]\rangle \Rightarrow \langle r_i : \mathcal{R}[\,[V_2/\mathtt{x}]M'\,]\rangle}\ app$$

$$\frac{V_1 = (\lambda \mathtt{x} : A\,.\,M') \quad V_2\ \mathtt{tvalue}}{\langle l_i : \mathcal{S}[\,V_1\ V_2\,]\rangle \Rightarrow \langle l_i : \mathcal{S}[\,[V_2/\mathtt{x}]M'\,]\rangle}\ app'$$

$$\frac{V\ \mathtt{tvalue}}{\langle r_j : V\rangle; \langle r_i : \mathcal{R}[\,r_j\,]\rangle \Rightarrow \langle r_j : V\rangle; \langle r_i : \mathcal{R}[\,V\,]\rangle}\ syncr$$

$$\frac{V\ \mathtt{tvalue}}{\langle r_j : V\rangle; \langle l_i : \mathcal{S}[\,r_j\,]\rangle \Rightarrow \langle r_j : V\rangle; \langle l_i : \mathcal{S}[\,V\,]\rangle}\ syncr'$$

The rules for function application are straightforward. Note that synchronization on a result label r_i may happen implicitly at any time, but it only becomes necessary when the structure of a value is observed. For example, synchronization is forced to occur before we may apply the *app* rule, because the *app* rule requires that V_1 have the form $\lambda \mathtt{x} : A \,.\, M'$.

$$\frac{V = \mathtt{box}\, M \quad r_j \ \mathtt{fresh}}{\langle r_i : \mathcal{R}[\,\mathtt{let\ box\,u} = V\ \mathtt{in}\, N\,]\rangle \ \Rightarrow\ \langle r_j : M\rangle; \langle r_i : \mathcal{R}[\,[[r_j/\mathtt{u}]]N\,]\rangle}\ \mathit{letbox}$$

$$\frac{V = \mathtt{box}\, M \quad r_j \ \mathtt{fresh}}{\langle l_i : \mathcal{S}[\,\mathtt{let\ box\,u} = V\ \mathtt{in}\, N\,]\rangle \ \Rightarrow\ \langle r_j : M\rangle; \langle l_i : \mathcal{S}[\,[[r_j/\mathtt{u}]]N\,]\rangle}\ \mathit{letbox}'$$

$$\frac{V = \mathtt{box}\, M \quad r_j \ \mathtt{fresh}}{\langle l_i : \mathtt{let\ box\,u} = V\ \mathtt{in}\, F\rangle \ \Rightarrow\ \langle r_j : M\rangle; \langle l_i : [[r_j/\mathtt{u}]]F\rangle}\ \mathit{letbox}_p$$

The *letbox* rules governing the use of $\Box A$ terms spawn an independent process for evaluation of the boxed term M. Meanwhile, the result label r_j is substituted for $\mathtt{u}$ in N. Label r_j will serve as a placeholder for the value of M, allowing us to achieve some concurrency in evaluation. The rule letbox_p defines the behavior of the variant in which the body F is an expression.

$$\frac{V = \mathtt{dia}\, E \quad l_j \ \mathtt{fresh}}{\langle l_i : \mathtt{let\ dia\,x} = V\ \mathtt{in}\, F\rangle \ \Rightarrow\ \langle l_j : \langle\langle E/\mathtt{x}\rangle\rangle F\rangle; \langle l_i : l_j\rangle}\ \mathit{letdia}$$

Finally, the *letdia* rule defines the way in which we may make use of a term $\Diamond A$. Recall that expressions proving $A\,\mathtt{poss}$ are evidence that A is true "somewhere". To respect this logical interpretation, we choose the particular world l_j we had in mind when we made the judgement $E \div A$. We then send the code F to be evaluated at that world (with E). The original process becomes a placeholder of the form $\langle l_i : l_j\rangle$, and we effectively shift our perspective to the world l_j. By performing the substitution $\langle\langle E/\mathtt{x}\rangle\rangle F$, expression E is used to validate the assumption $\mathtt{x} : A$ on which F depends. The substitution is well-defined, both E and F being expressions.[6] Note that no synchronization on l_j is permitted; such a synchronization principle would be an unsafe, non-logical operation, corresponding to the (invalid) deduction $A\,\mathtt{poss} \vdash A\,\mathtt{true}$.

It may seem strange that both the box and diamond constructs involve making an arbitrary choice of where to run the enclosed term or expression. In fact, for diamond elimination, the choice is not arbitrary. The requirement that l_j `fresh` is an approximation of the true, correct behavior. As noted above, we should evaluate $E \div A$ at the particular world dictated by its typing. Unfortunately, the proof language does not capture the particular world we

[6]Here we make use of the fact that location labels l_i are *not* expressions, hence $E \neq l_i$.

had in mind when making the judgement $E \div A$. In a lower-level language with explicit representation of worlds, it is likely possible to give a more precise account of where particular expressions should be evaluated.

5 Properties of the Semantics

5.1 Substitution Properties

With some trivial generalization, namely the addition of a label typing context Λ, the following substitution properties from [8] hold:

$$\begin{array}{rcrcl}
\Lambda;\Delta;\Gamma,\mathtt{x}:B,\Gamma' \vdash N : A & \wedge & \Lambda;\Delta;\Gamma \vdash M : B & \Longrightarrow & \Lambda;\Delta;\Gamma,\Gamma' \vdash [M/\mathtt{x}]N : A \\
\Lambda;\Delta;\Gamma,\mathtt{x}:B,\Gamma' \vdash F \div A & \wedge & \Lambda;\Delta;\Gamma \vdash M : B & \Longrightarrow & \Lambda;\Delta;\Gamma,\Gamma' \vdash [M/\mathtt{x}]F \div A \\
\Lambda;\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash N : A & \wedge & \Lambda;\Delta;\cdot \vdash M : B & \Longrightarrow & \Lambda;\Delta,\Delta';\Gamma \vdash [[M/\mathtt{u}]]N : A \\
\Lambda;\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash F \div A & \wedge & \Lambda;\Delta;\cdot \vdash M : B & \Longrightarrow & \Lambda;\Delta,\Delta';\Gamma \vdash [[M/\mathtt{u}]]F \div A \\
\Lambda;\Delta;\mathtt{x}:B \vdash F \div A & \wedge & \Lambda;\Delta;\Gamma \vdash E \div B & \Longrightarrow & \Lambda;\Delta;\Gamma \vdash \langle\langle E/\mathtt{x}\rangle\rangle F \div A
\end{array}$$

Generally speaking, the substitution properties continue to hold in the presence of result labels because labels are closed values, insensitive to the contents of Δ and Γ. Note that we may use weakening on label contexts Λ when required to achieve matching contexts.

5.2 Evaluation Context Typing

We must now consider typing of terms and expressions of the form $\mathcal{R}[\,M\,]$ (or $\mathcal{S}[\,M\,]$). Although both forms simply denote a term (or expression), it is possible to say more about the typing of the term used to fill the "hole" in the context. A key property of evaluation contexts, as they have been defined, is that we never evaluate below a binding construct. Hence we know that the term filling the hole will be typed in the same combined context $\Lambda;\Delta;\Gamma$ as the surrounding parts of the term (or expression). The following inversion principles apply:

$$\begin{array}{lrcl}
(1) & \Lambda;\Delta;\Gamma \vdash \mathcal{R}[\,M\,] : A & \Longrightarrow & \Lambda;\Delta;\Gamma \vdash M : B \\
(2) & \Lambda;\Delta;\Gamma \vdash \mathcal{S}[\,M\,] \div A & \Longrightarrow & \Lambda;\Delta;\Gamma \vdash M : B \\
(3) & \Lambda;\Delta;\Gamma \vdash \mathcal{S}[\,E\,] \div A & \Longrightarrow & \Lambda;\Delta;\Gamma \vdash E \div A
\end{array}$$

Proof:

(1) By induction on the form of $\mathcal{R}$. In the base case $\mathcal{R} = []$, $\mathcal{R}[\,M\,] = M$ so we are done. In the non-trivial cases, we apply inversion for the $\rightarrow E$ or $\Box E$ typing rules, obtaining a smaller term $\mathcal{R}'[\,M\,]$, which must be typed under the *same* assumptions $\Lambda;\Delta;\Gamma$.

(2,3) Given an expression of the form $\mathcal{S}[\,E\,] \div A$, we know $\mathcal{S}$ must have the form $[]$, so (3) is trivially true. Given an expression of the form $\mathcal{S}[\,M\,] \div A$,

either $\mathcal{S} = \{\mathcal{R}\}$, in which case we appeal to (1) immediately, or we can apply inversion for the $\Box E_p$ or $\Diamond E$ typing rules obtaining a context $\mathcal{R}[\,M\,]$ on which (1) will apply.

In particular, this means that if we assume $\mathcal{S}[\,M\,]$ is closed (with respect to Δ and Γ), then M is closed as well. By analogy with a stack, it would also be possible to give a frame-by-frame account of the typing of $\mathcal{S}[\,M\,]$, in which we specify the form of type B which is required of M.

5.3 Type Preservation

The operational semantics preserves process typing. As the process configuration evolves, new processes are created, but existing processes remain well-typed (at the same type). Type preservation for the semantics is stated as:

$$\vdash_C C : \Lambda \quad \wedge \quad C \Rightarrow C' \quad \Longrightarrow \quad \exists\, \Lambda' \supseteq \Lambda\,.\ \vdash_C C' : \Lambda'$$

Proof: First note that each transition affects only one or two processes in the configuration. We may focus our attention on those processes only; if type preservation holds locally, and we can show $\Lambda' \supseteq \Lambda$, then type preservation will hold globally. The proof proceeds by cases on the $C \Rightarrow C'$ judgement.

app We may assume process $\langle r_i : \mathcal{R}[\,V_1\ V_2\,]\rangle$ is well-typed, in the sense that $\Lambda_1; \cdot; \cdot \vdash \mathcal{R}[\,V_1\ V_2\,] : C$. Note that $V_1 = \lambda \mathtt{x} : A\,.\,M'$. By evaluation context typing, we know $\Lambda_1; \cdot; \cdot \vdash V_1\ V_2 : B$. Obviously, $V_1\ V_2$ is syntactically a function application. Therefore, we apply inversion to obtain $\Lambda_1; \cdot; \mathtt{x} : A \vdash M' : B$ and $\Lambda_1; \cdot; \cdot \vdash V_2 : A$. By the ordinary substitution property, $\Lambda_1; \cdot; \cdot \vdash [V_2/\mathtt{x}]M' : B$. No processes are spawned, so $\Lambda' = \Lambda$.

app′ Similar to *app*, though in an expression context.

syncr By assumption, $\langle r_j : V\rangle; \langle r_i : \mathcal{R}[\,r_j\,]\rangle \Rightarrow \langle r_j : V\rangle; \langle r_i : \mathcal{R}[\,V\,]\rangle$. We may also assume that both processes are well-formed, $\Lambda_1; \cdot; \cdot \vdash V : A$ and $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash \mathcal{R}[\,r_j\,] : B$. By weakening (w.r.t. Λ) we may replace r_j with V in $\mathcal{R}[\,r_j\,]$. Hence we conclude $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash \mathcal{R}[\,V\,] : B$. Also note that $\Lambda' = \Lambda$.

syncr′ Similar to *syncr*, though in an expression context.

letbox We assume process $\langle r_i : \mathcal{R}[\,\mathtt{let\ box\,u = }V\ \mathtt{in}\,N\,]\rangle$ is well-typed. By assumption, $V = \mathtt{box}\,M$. By evaluation context typing and inversion on $\Box E$ and $\Box I$ rules, we know that $\Lambda_1; \cdot; \cdot \vdash M : A$. Hence the new process $\langle r_j : M\rangle$ is well-typed. By result label typing, $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash r_j : A$. By the properties of substitution, $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash [[r_j/\mathtt{u}]]N : B$. Hence the original process remains well-typed. We have extended the process typing, so $\Lambda' \supset \Lambda$.

letbox′ Similar to *letbox*, though in an expression context.

letbox$_p$ We may assume process $\langle l_i : \texttt{let box u} = V \texttt{ in } F\rangle$ is well-typed, that is, $\Lambda_1; \cdot; \cdot \vdash \texttt{let box u} = V \texttt{ in } F \div B$. Also by assumption, $V = \texttt{box}\, M$. We apply inversion on the typing rules $\Box E_p$ and $\Box I$ to obtain $\Lambda_1; \cdot; \cdot \vdash M : A$. Hence the new process $\langle r_j : M\rangle$ is well-typed. By result label typing, $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash r_j : A$. By the properties of substitution, $\Lambda_1, r_j :: A, \Lambda_2; \cdot; \cdot \vdash [[r_j/\mathtt{u}]]F \div B$. Hence the original process remains well-typed. We have extended the process typing, so $\Lambda' \supset \Lambda$.

letdia We assume process $\langle l_i : \texttt{let dia x} = V \texttt{ in } F\rangle$ is well-typed. Also by assumption, $V = \texttt{dia}\, E$. We may apply inversion on typing to obtain $\Lambda_1; \cdot; \cdot \vdash E \div A$ and $\Lambda_1; \cdot; \mathtt{x} : A \vdash F \div B$. By a substitution property, $\Lambda_1; \cdot; \cdot \vdash \langle\langle E/\mathtt{x}\rangle\rangle F \div B$, so the new process is well-formed. The original process remains well-formed by the rule for typing processes of the form $\langle l_i : l_j\rangle$. We have extended the process typing, so $\Lambda' \supset \Lambda$.

5.4 Progress

A progress property for the semantics ensures that well-typed process configurations do not get stuck in an erroneous, non-value, state. For purposes of the progress theorem, we consider processes of the form $\langle l_j : l_i\rangle$ to be in an acceptable terminal state. A process configuration C is terminal iff all its processes are terminal.

$$\frac{}{\langle l_j : l_i\rangle \texttt{ terminal}} \qquad \frac{V \texttt{ evalue}}{\langle l_i : V\rangle \texttt{ terminal}} \qquad \frac{V \texttt{ tvalue}}{\langle r_i : V\rangle \texttt{ terminal}}$$

A progress property for the semantics may be stated as follows:

$$\vdash_C C : \Lambda \quad \Longrightarrow \quad C \texttt{ terminal} \quad \vee \quad \exists C' \,.\, C \Rightarrow C'$$

Proof: If all processes are terminal, the statement is true. Assume this is not the case. Then there is a process $\langle r_i : M\rangle$ in which M is not a term value or a process $\langle l_i : E\rangle$ in which E is not an expression value. We may assume that r_i (or l_i) is chosen to be the least such label in the order determined by $\vdash_C C : \Lambda$; that is, all r_j (or l_j) prior to r_i (or l_i) refer to processes in terminal form. By the definition of configuration typing, we can infer $\Lambda; \cdot; \cdot \vdash M : A$ (or $\Lambda; \cdot; \cdot \vdash E \div A$). Now it will be sufficient to show that either progress can be made on $\langle r_i : M\rangle$ (alternately $\langle l_i : E\rangle$) or M is a term value (E is an expression value) violating the original assumption that the process is non-terminal. Hence

the following lemma is sufficient for progress:

$$
\begin{array}{lll}
(1) & \vdash_C C : \Lambda & \\
& \wedge \quad C \ \texttt{terminal} & \\
& \wedge \quad \Lambda; \cdot; \cdot \vdash M : A \quad \Longrightarrow & \exists C'.\exists M' \ . \ C; \langle r_i : M \rangle \Rightarrow C'; \langle r_i : M' \rangle \\
& & \vee \quad M \ \texttt{tvalue} \\
\text{and } (2) & \vdash_C C : \Lambda & \\
& \wedge \quad C \ \texttt{terminal} & \\
& \wedge \quad \Lambda; \cdot; \cdot \vdash E \div A \quad \Longrightarrow & \exists C'.\exists E' \ . \ C; \langle l_i : E \rangle \Rightarrow C'; \langle l_i : E' \rangle \\
& & \vee \quad \exists C'.\exists l_j \ . \ C; \langle l_i : E \rangle \Rightarrow C'; \langle l_i : l_j \rangle \\
& & \vee \quad E \ \texttt{evalue}
\end{array}
$$

The proof is by induction on term and expression typing:

res $M = r_i$ is a value (though we could also make progress by synchronizing).

hyp* $M = \mathtt{u}$. Vacuously true, since $\mathtt{u}$ is not a closed term.

hyp $M = \mathtt{x}$. Vacuously true, since $\mathtt{x}$ is not a closed term.

$\rightarrow I$ $M = \lambda \mathtt{x} : A \,.\, M'$ is a value.

$\rightarrow E$ $M = M_1 \ M_2$. By inversion on typing we know $\Lambda; \cdot; \cdot \vdash M_1 : A \rightarrow B$ and $\Lambda; \cdot; \cdot \vdash M_2 : A$. We can then apply the induction hypothesis to the two subterms, concluding that (1) progress can be made on M_1 or M_2 (in suitable evaluation contexts) or (2) both M_1 and M_2 are values. If progress can be made on either, we are done. If both M_1,M_2 are values, then we know by canonical forms that $M_1 = \lambda \mathtt{x} : A \,.\, M'$ or $M_1 = r_j$. In the former case, we can apply rule *app*; in the latter case we can make progress by synchronizing because we have assumed that all processes in C are terminal (and hence values).

$\Box E$ $M = \mathtt{let\ box\,u = } M' \mathtt{\ in\ } N'$. We also know (by inversion) that $\Lambda; \cdot; \cdot \vdash M' : \Box A$. If M' is not a value, we make progress inductively (in a context $\mathcal{R}[\,M'\,]$, for suitable $\mathcal{R}$). If M' is a value, then by canonical forms it must be either $M' = r_j$ or $M' = \mathtt{box}\, M''$. If $M' = r_j$ then we may make progress by synchronizing. If $M' = \mathtt{box}\, M''$ then we may apply rule *letbox*.

$\Box E_p$ $E = \mathtt{let\ box\,u = } M' \mathtt{\ in\ } F$. Similar to the previous case, except if M' is a value of the form $\mathtt{box}\, M''$, we make progress through rule $letbox_p$.

poss $E = \{M\}$. By inversion on typing, $\Lambda; \cdot; \cdot \vdash M : A$. By the induction hypothesis, we can conclude either progress is possible on M (in a suitable context $\mathcal{R}$), or M $\mathtt{tvalue}$. In the former case, progress is also possible on $\{M\}$ (in context $\mathcal{S} = \{\mathcal{R}\}$). In the latter case, we conclude $\{M\}$ $\mathtt{evalue}$.

$\Diamond E$ $E = \mathtt{let\ dia\,x = } M \mathtt{\ in\ } F$. We also know (by inversion) that $\Lambda; \cdot; \cdot \vdash M : \Diamond A$. If M is not a value, we make progress inductively in context $\mathcal{S}[\,M\,]$ for suitable $\mathcal{S}$. If M is a value, then by canonical forms it must be either

$M = r_j$ or $M = \texttt{dia}\, E$. If $M = r_j$ then we may make progress by synchronizing. If $M = \texttt{dia}\, E$ then we may apply rule *letdia*, producing a terminal process $\langle l_i : l_j \rangle$.

6 Why Modal Types?

As noted earlier, the typing rules for $\texttt{box}\, M$ and $\texttt{let dia x} = M\, \texttt{in}\, F$, impose restrictions on the occurrence of variables within M and F respectively. Our project of demonstrating that modal logic naturally describes distributed computation will not have been entirely successful unless we can justify these restrictions (and others). Why should we treat variables with suspicion? Why not, for example, allow arbitrary mobility of code, supported by a mechanism which can marshal closures under arbitrary environments? Why not permit synchronization on location labels, bringing back to the spawning process the value computed by an expression?

Unfortunately, in the presence of truly localized resources, the preceding suggestions will fail. There may be certain localized values (such as heap addresses) or resources (hardware devices) which do not make sense at any other world. Safely marshalling such localized values would be impossible, so we must *insist* on these restrictions!

More fundamentally, the laws of modal logic are *designed* to characterize models in which truth is relative to an underlying set of worlds. This includes the possibility that certain propositions are true in some worlds but not in others. In this way, localized resources arise inevitably from the logical definitions. If this was not clear from the beginning, it is because the simplified proof language we work with in this paper is somewhat underdeveloped with respect to primitive expressions ($E \div A$). The core language only allows formulation of proofs (programs) which are generally valid in *any* model of modal logic. Additional primitive expressions impose more structure on the model, corresponding to the presence of particular local resources in particular locations.

As we argued above, there are reasons to believe that the language of modal logic is generally useful for writing programs which respect the locality of certain resources. In this light, it is clear that the structure and behavior of expressions is interesting in its own right, even in the absence of additional primitive expressions. However, if one decides that programming with localized resources is not necessary to support the sort of applications one has in mind, then it is not mandatory to abide by these restrictions. Modal logic is not the proper way to characterize systems in which everything is potentially mobile, nor can modal logic be applied in domains for which there is no stable notion of "truth" relative to locations.

However, modal logic is *universal* in the following sense: Any safe language which claims to describe the behavior of and access to localized resources must take similar precautions to prevent these localized entities from escaping the context in which they are defined. Many ad-hoc solutions to this problem have been devised. One typical approach is to make various restrictions on the types

of values which may be marshalled, only permitting mobility for certain "primitive" types. Another is to cover up the existence of localized entities, making a heroic effort to "copy everything" (even if this leads to semantic anomalies). We hope this work makes it clear that it is *not* the "primitiveness" of some value types or the amount of effort one wants to put into developing a language runtime which should limit mobility of certain code and values. Mobility can be understood purely on logical grounds. The localized expressions are simply the things which are evidence for $A\,\mathtt{poss}$ (A is true at some particular world). If one admits at that such entities exist, then they *must* behave logically and operationally as dictated by the language of modal logic.

7 Practical Programming with $\Box$ & $\Diamond$

We must keep in mind that there are two kinds of reason to program with modal types $\Box A$ and $\Diamond A$ — safety (logical consistency) and achieving concurrency (by moving terms elsewhere for further evaluation).

From a logical point of view, $\mathtt{box}\,M$, $\mathtt{dia}\,E$ and their elimination forms provide a safe way to work with mobile code and localized resources. In giving the operational semantics, we restricted our attention to closed programs (closed with respect to both Δ and Γ). However, it is also reasonable to consider programming in some initial environment $\Delta_0; \Gamma_0$. For our purposes, valid hypotheses (in Δ_0) are the most useful. Hypotheses of the type $\Diamond A$ can be used to represent a fixed set of local resources (of which we are aware *a priori*).

On the other hand, adopting a behavioral point of view, the use of $\Box A$ can have the side effect of introducing concurrency. This may also be important to the programmer, indeed, achieving concurrent evaluation may be the primary goal when using $\Box A$. Mobility is somewhat intertwined with concurrency because we assume each abstract "location" has the capability to compute independently of the others. However, this is really a secondary effect of the logical/spatial interpretation.

7.1 Definition of Recursion

Many interesting programs require recursion to specify. These programs can be characterized as having a variable degree of parallelism. That is, they may "unroll" at runtime to a tree-structured computation, or any other form of computation involving an unbounded number of worlds. To support recursion, we add add the following fixpoint operators to the language, with typing as follows:

$$\frac{\Delta; \Gamma, \mathtt{x} : A \vdash M : A}{\Delta; \Gamma \vdash \mathtt{fix}\,(\mathtt{x} : A)\,.\,M : A}\ fix$$

$$\frac{\Delta, \mathtt{u} :: A; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M : A}\ fix_v$$

One may also consider recursion over expressions, but it is not clear how this would be practically useful, since expressions are localized. Clearly, the addition of $\mathtt{fix}\,(\mathtt{x} : A)\,.\,M$ disturbs the logical properties of the language, since $\vdash \mathtt{fix}\,(\mathtt{x} : A)\,.\,x : A$ for any type A. The usual caveats about recursion apply, namely that ill-founded "proofs" of this sort will not terminate under evaluation. At first it might seem that fix_v is a redundant derivable rule. Indeed, it is possible to provide a definition for fix_v as a proof schema:

$$\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M \quad \equiv \quad \mathtt{fix}\,(\mathtt{y} : \Box A)\,.\,\mathtt{let\ box\,u = y\ in}\,(\mathtt{box}\,M)$$

However, when one considers the behavior of such terms under evaluation, it becomes clear that this is not a desirable way to define recursion over valid terms. For example, the simple fixpoint $\mathtt{fix}_v\,(\mathtt{u} :: A \to A)\,.\,\lambda\mathtt{x} : A\,.\,M$ would never terminate. The problem is that the definition is too eager in unwinding the recursion. Hence we must extend the operational semantics for each flavor of recursion, defining it in such a way that the unwinding is performed lazily.

$$\frac{}{\langle W : \mathcal{S}[\,\mathtt{fix}\,(\mathtt{x} : A)\,.\,M\,]\rangle \;\Rightarrow\; \langle W : \mathcal{S}[\,[\mathtt{fix}\,(\mathtt{x} : A)\,.\,M/\mathtt{x}]M\,]\rangle}\ fix$$

$$\frac{}{\langle W : \mathcal{S}[\,\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M\,]\rangle \;\Rightarrow\; \langle W : \mathcal{S}[\,[[\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]]M\,]\rangle}\ fix_v$$

Type preservation and progress proofs for the operational semantics can be extended to account for fixpoint. In the case of the progress theorem, we note that fixpoint is not a value, but that we can always apply one of the rules fix or fix_v. In the case of the type preservation theorem, the substitution property (a term for an ordinary or valid variable) will ensure proper typing of the result.

7.2 Axioms of Modal Logic

Below are reproduced the axioms of S4, together with their realizations as proof terms:

$\vdash$ $S \equiv \lambda\mathtt{x} : A \to B \to C\,.\,\lambda\mathtt{y} : A \to B\,.\,\lambda\mathtt{z} : A\,.\,(\mathtt{x\ z})(\mathtt{y\ z})$
: $((A \to B \to C) \to (A \to B) \to A \to C)$
$\vdash$ $K \equiv \lambda\mathtt{x} : A\,.\,\lambda\mathtt{y} : B\,.\,\mathtt{x}$
: $(A \to (B \to A))$
$\vdash$ $DB \equiv \lambda\mathtt{x} : \Box(A \to B)\,.\,\lambda\mathtt{y} : \Box A\,.\,\mathtt{let\ box\,u = x\ in}\,(\mathtt{let\ box\,v = y\ in box}\,(\mathtt{u\ v}))$
: $\Box(A \to B) \to (\Box A \to \Box B)$
$\vdash$ $RB \equiv \lambda\mathtt{x} : \Box A\,.\,\mathtt{let\ box\,u = x\ in\,u}$
: $(\Box A \to A)$
$\vdash$ $S4 \equiv \lambda\mathtt{x} : \Box A\,.\,\mathtt{let\ box\,u = x\ in\ box\,box\,u}$
: $(\Box A \to \Box\Box A)$
$\vdash$ $RD \equiv \lambda\mathtt{x} : A\,.\,\mathtt{dia}\{\mathtt{x}\}$
: $(A \to \Diamond A)$
$\vdash$ $TD \equiv \lambda\mathtt{x} : \Diamond\Diamond A\,.\,\mathtt{dia}\,(\mathtt{let\ dia\,y = x\ in}\,(\mathtt{let\ dia\,z = y\ in}\,\{\mathtt{z}\}))$
: $(\Diamond\Diamond A \to \Diamond A)$
$\vdash$ $DD \equiv \lambda\mathtt{x} : \Box(A \to B)\,.\,\mathtt{let\ box\,u = x\ in}\,(\lambda\mathtt{y} : \Diamond A\,.\,\mathtt{dia}\,(\mathtt{let\ dia\,z = y\ in}\,\{\mathtt{u\ z}\}))$
: $\Box(A \to B) \to (\Diamond A \to \Diamond B)$

These axioms can be generally useful in implementing various common patterns of distributed computation. For example, axiom RB shows us how to access a boxed, global resource "here", and DD shows us how to make use of a global resource in combination with a localized resource.

From a behavioral perspective, we may compose axioms DB and RB to spawn a function application (`u v`). The axioms relating to $\Diamond$ do not exhibit behavior immediately, because `dia`E is a value encapsulating a localized expression or computation. However, if we apply diamond elimination to these values they may have interesting effects. Axiom TD, for example, allows us to shift perspective to a location two "hops" away to evaluate an expression.

7.3 Example (Local Resources)

As an example in which we may be forced to use local resources to solve a problem, consider the problem of interaction with a user at a particular "home" console. Assuming `console` :: $\Diamond$`con` represents such a localized resource, we may program as follows:

```
let dia c = console in
        write c ``Enter a number:'';
        write c ``answer = '';
        write c ((λ x : int . M) (read c))
```

At runtime, the body of the let-expression is evaluated at the location of `console`, producing output on the user's terminal.

The following solution would also be well-typed, but it delays the interactive computation, by encapsulating it inside of a dia constructor.

```
dia
let dia c = console in
        write c ``Enter a number:'';
        write c ``answer = '';
        write c ((λ x : int . M) (read c))
```

To perform the same computation repeatedly, we use the fixpoint operator. Note that we must use the closed, valid form of recursion $\texttt{fix}_v\,(\texttt{u} :: A)\,.\,M$ because the entire loop is required to be mobile. However, we will not take full advantage of this mobility, since each instance of the loop body will be evaluated at the same location (the console location).

```
fix_v loop :: ◇ unit .
dia
let dia c = console in
        write c ``Enter a number:'';
        write c ``answer = '';
        write c ((λ x : int . M) (read c));
        let dia l = loop in l
```

Note the form (`let dia l = loop in l`) of invocation required to produce additional loop iterations. Essentially, this removes the `dia` constructor from the freshly unrolled copy of the loop, resuming execution with the enclosed expression.

7.4 Example (Local Resources of Another Kind)

Though the previous example made use of a resource `console`, tied *a priori* to a particular location, it is also possible to model a situation in which values or effects of a computation are localized, though the code of the program (as written) is not. Consider, for example, the case of reference cells. We could provide access to these storage locations through the following set of primitives:

$$\frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash \mathtt{ref}\, M \div \mathtt{ref}\, A}\; refI \qquad \frac{\Delta;\Gamma \vdash M : \mathtt{ref}\, A}{\Delta;\Gamma \vdash\, !\, M : A}\; refE$$

$$\frac{\Delta;\Gamma \vdash M : \mathtt{ref}\, A \quad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M := N : \mathtt{unit}}\; refSet$$

Note that the $\mathtt{ref}\, M$ syntax is globally available. However, we choose to make $\mathtt{ref}\, M$ an expression, not a term. The intuition behind this choice is that $\mathtt{ref}\, M$ will have a localized effect during evaluation, and its meaning will become dependent on the particular location at which the effect occurs. Specifically, $\mathtt{ref}\, M$ will evaluate to a heap address a, with typing $a \div \mathtt{ref}\, A$ (addresses will be the canonical form for expressions of type $E \div \mathtt{ref}\, A$). The intuition behind the elimination rule is that we may retrieve the value of a reference if that reference is available "here". Similarly, in order to update the contents of a reference cell, we must have $\mathtt{ref}\, A$ available locally.

Effectively, the typing principles given above force us to use references in a monadic style, encapsulating expressions which allocate and manipulate references with $\Diamond$. The form of $\Diamond E$ only allows us to work with one reference cell at a time; this precludes any conflict over two resources available at different locations. With product types it is possible to work around this restriction to some extent, by storing more than one value in a single reference cell.

References, being expressions, are protected by typing rules governing possibility. For example, we *cannot* make a reference mobile:

```
let box u = box (ref 0) in          (* ill-typed *)
        u := (!u + 1)
```

The only way to make the reference mobile (in an indirect sense) is to encapsulate it as $\mathtt{dia}(\mathtt{ref}\, 0)$. In this way, we can move the *term* $\mathtt{dia}(\mathtt{ref}\, 0)$ anywhere, though the effect of the computation (allocation of a reference cell) is delayed until $\mathtt{ref}\, 0$ is revealed by diamond elimination and a particular location is chosen.

```
let box u = box (dia (ref 0)) in
let dia x = u in                (* allocate ref cell *)
        x := (!x + 1);
        let dia y = u in        (* a new ref cell, y ≠ x *)
                y := (!y + 1);
                y               (* attempt to ''return'' y *)
```

Although u is available at many locations, as `dia (ref 0)`, the operational semantics of $\Diamond I$ and $\Diamond E$ dictate that `x` and `y` are distinct reference cells. Note also that the value of `y` cannot "escape" the location at which it is allocated, because the typing rules prevent us from relying on the results of evaluating an expression. The occurrence of `y` in the final line of the program is perfectly well-typed, but will not have any meaningful effect.

7.5 Example (Concurrency)

Finally, consider a program for computing the nth Fibonacci number recursively. Additionally, we would like to have each recursive call evaluated at a different world, achieving a degree of concurrency by distributing the work. A basic implementation of `fib` is given below:

```
fix fib : int → int .
λ n : int .
        if (n < 2) then n
        else (fib (n-1)) + (fib (n-2))
```

This term is well-typed, having type $int \to int$. It does not, however, exhibit the desired parallelism. To achieve the sort of arbitrary mobility that will allow each recursive call to be evaluated independently, it is clear we should look to `box` and `let box u = M in N`. We will have to decorate the type of `fib` with $\Box$ to achieve the proper effect. One way to achieve distributed evaluation is as follows:

```
fix_v fib :: □int → int .
λ n : □int .
        let box u = n in
                if (u < 2) then u
                else
                        let box a = box (fib (box (u - 1))) in
                        let box b = box (fib (box (u - 2))) in
                        a + b
```

This realization of `fib` is at type $\Box int \to int$. Note that it is necessary to use recursion over valid terms $(\texttt{fix}_v\,(\texttt{u} :: A)\,.\,M)$ because we want `fib` to be available at any world we see fit to spawn $(\texttt{box}\,(\texttt{fib}\,(\texttt{box}\,(u-1))))$. Now when `fib` is applied to a boxed integer (`fib (box 2)`), the process configuration evolves as follows:

$$
\begin{array}{ll}
 & \langle r_0 : \texttt{fib}\ (\texttt{box}\, 2)\rangle \\
\Rightarrow^{*} & \langle r_0 : \texttt{let box u = box 2 in} \ldots\rangle \\
\Rightarrow & \langle r_0 : \texttt{if}\ \ (r_1 < 2)\ldots\rangle; \langle r_1 : 2\rangle \\
\Rightarrow^{*} & \langle r_0 : r_2 + r_3\rangle; \langle r_1 : 2\rangle; \langle r_2 : \texttt{fib box}\,(r_1 - 1)\rangle; \langle r_3 : \texttt{fib box}\,(r_1 - 2)\rangle \\
\Rightarrow^{*} & \langle r_0 : r_2 + r_3\rangle; \langle r_1 : 2\rangle; \langle r_2 : \texttt{if}\ \ (r_4 < 2)\ldots\rangle; \langle r_3 : \texttt{if}\ \ (r_5 < 2)\ldots\rangle; \langle r_4 : r_1 - 1\rangle; \langle r_5 : r_1 - 2\rangle \\
\Rightarrow^{*} & \langle r_0 : r_2 + r_3\rangle; \langle r_1 : 2\rangle; \langle r_2 : \texttt{if}\ \ (r_4 < 2)\ldots\rangle; \langle r_3 : \texttt{if}\ \ (r_5 < 2)\ldots\rangle; \langle r_4 : 1\rangle; \langle r_5 : 0\rangle \\
\Rightarrow^{*} & \langle r_0 : r_2 + r_3\rangle; \langle r_1 : 2\rangle; \langle r_2 : r_4\rangle; \langle r_3 : r_5\rangle; \langle r_4 : 1\rangle; \langle r_5 : 0\rangle \\
\Rightarrow^{*} & \langle r_0 : r_2 + r_3\rangle; \langle r_1 : 2\rangle; \langle r_2 : 1\rangle; \langle r_3 : 0\rangle; \langle r_4 : 1\rangle; \langle r_5 : 0\rangle \\
\Rightarrow^{*} & \langle r_0 : 1 + 0\rangle; \langle r_1 : 2\rangle; \langle r_2 : 1\rangle; \langle r_3 : 0\rangle; \langle r_4 : 1\rangle; \langle r_5 : 0\rangle \\
\Rightarrow & \langle r_0 : 1\rangle; \langle r_1 : 2\rangle; \langle r_2 : 1\rangle; \langle r_3 : 0\rangle; \langle r_4 : 1\rangle; \langle r_5 : 0\rangle
\end{array}
$$

Note that the pattern `let box a = box (fib ...) in ...` is used to spawn two applications of `fib` for concurrent evaluation. The results of both branches must be received (with the *syncr* rule) before evaluation of `(a + b)` can proceed.

8 Related Work

Most prior and ongoing related work seems to have taken, as a starting point, a general process calculus, such as the Pi or join calculus. These sorts of calculi model the connectivity of processes, but not location and localized resources. They also typically permit arbitrary and dynamically changing communication patterns between processes. Starting from such a calculus, a notion of location is added to the operational semantics, the language is extended with one or more primitives for mobility, and (optionally) restrictions are imposed on how and where a process may safely move. The authors mentioned below have dealt with many of the same issues that we address in this paper. But because they start from a behavioral, process-oriented perspective, the issues and problems present themselves in slightly different forms, so the solutions they arrive at are naturally different.

However, on a deep level, both our work and theirs represent an attempt to impose some logical structure on mobility and distributed computation. We believe the critical questions are these: What are the local resources that distinguish locations from one another? And where may fragments of code (which might depend on these resources) move to and from safely?

In our work, the local resources are expressions, and dependency on such local resources is through variables (locally true hypotheses). In the related work based on process calculi, channel and/or location names usually play both roles simultaneously. However, it is crucial to note that process calculi typically allow changing the scope of channel names (via a scope extrusion rule). This makes it quite challenging to come up with a stable, coherent notion of what names are in scope at each location. Hence mechanisms for enforcing or characterizing the locality of channel names or processes can become quite complex.

8.1 Mobile Ambients

Mobile ambients, as developed by Cardelli and Gordon [3], are a novel way of adding locality to a process calculus. The ambient notation $n[\ldots]$ allows representation of location in process configurations. Simultaneously, ambients also facilitate communication by providing a space in which processes may exchange messages (eliminating the need for primitive channels).

In subsequent papers [5, 1, 2], an "ambient logic" is developed to characterize the behavior and spatial distribution of processes. Ambient logic is not intended to be a system for assigning types to processes; it is a language for making statements about a given process configuration (considered as a model for the logic). These propositions are then either satisfied by the given model, or not, according to the definitions of the logic. The logic includes modal operators of both the spatial and temporal variety which are interpreted by reference to spatial (hierarchical inclusion) and temporal (reduction steps) notions of accessibility. The full ambient logic is very precise, and allows one to specify undecidable properties of a program. Verifying a formula in decidable fragments of ambient logic is typically accomplished by model-checking.

Notably, Cardelli and Gordon (in [4]) have extended ambient logic with propositions expressing hiding, revelation, and freshness of names in order to characterize the scope and mobility of names. However, since processes in the ambient calculus are not inherently required to preserve "locality" of names, most name-hiding properties must be formulated and proved (by model-checking) relative to a particular implementation.

8.2 DPI and Process Typing

Hennessy, *et. al.* have developed another language for distributed computation based on the Pi-calculus, called DPI. It extends the Pi-calculus with a notation for process location, and a simple $\texttt{go}\ l\ .\ P$ action which moves P to location l where execution of P resumes.

The typing systems developed for this language are described in papers by Hennessy, Riely, Yoshida, and others. [7, 9, 6] Their work is very closely related to the contents of this report, seeking to achieve the same goals, but in the context of a process calculus.

Their typing system restricts the scope of names so that processes in a location l are only allowed to access names declared in l. Names may escape the scope of their declaration, but only as "existential" values $n@l$, tagged with the location in which they are valid. In this manner, the authors achieve a stable notion of which resources are available at which locations. In fact, locations are characterized by "location types" $\texttt{loc}\{u_1 : A, u_2 : B, \ldots\}$, which effectively internalize the set of bound names in scope at that location. The authors also permit subtyping on location types which is similar to record subtyping.

Informally speaking, we can find counterparts to some modal types in the scheme of location types. For example, a term of type $\Box A$ corresponds to a process P which is well-formed in a location of type $\texttt{loc}\{\}$ (the top type of the

location typing hierarchy). Such a process may move to any location, since it depends on no local names. General terms of type $\Diamond A$ do not have a direct analogue in the DPI typing system, since processes cannot be removed from the context in which they are well-formed, but for the special case of channel names, $\Diamond A$ corresponds to the use of existential types $A@L$ (there exists a location L in which A) to characterize channel names which "escape" their original scope of definition.

Interestingly, the $\texttt{let dia x} = M \texttt{ in} F$ construct defined in this paper is quite similar to $\texttt{go}\ l\ .\ P$, in the sense that F (and P) are being sent to a new location. The difference is that $\texttt{let dia x} = \texttt{dia} E \texttt{ in} F$ allows F access only to the *value* of E, rather than all the resources in scope at E's location. This is a natural outcome, given that our language is oriented toward evaluation rather than interaction.

9 Conclusion and Future Work

Starting from an intuitionistic formulation of modal logic, we considered the proof terms for that logic as a programming language. We then developed an operational semantics which realized the desired interpretation, that worlds should correspond to locations where evaluation may occur. We have shown that the modal types $\Box A$ and $\Diamond A$ are a safe and natural way to utilize both global and local resources in a distributed computation. We have also argued that restrictions on the scope of variables are essential to ensure safe mobility in the presence of localized resources.

In future work, we plan to separate concurrency from distributed computation, hopefully in a way that preserves the logical structure of programs and their types. A separate notion of concurrency leads to the possibility of collocation of processes, requiring an operational model in which locations are separate from the notion of a process.

We also plan to investigate languages derived from an explicit-worlds formulation of modal logic, in which worlds and connections (witnesses for accessibility) are embedded in types and terms. This should make it possible to give a more fully-developed treatment of primitive expressions (representing local resources). Additionally, making worlds explicit will allow us give a more precise operational semantics for possibility, because terms of type $\Diamond A$ will be explicitly labeled with the world at which they are valid.

We also believe it would be possible to address many interesting new questions in the explicit framework, such as the role of assumptions about network topology in the construction of programs, the possibility of identifying and constraining communication patterns in distributed programs, and limiting resource usage (in the coarse sense of the number of worlds participating in evaluation). We also hope to find more examples and applications in which the modal typing discipline, while not logically mandatory, would have additional practical benefits, such as a simpler or more efficient implementation of mobility.

References

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

[2] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part II). In *CONCUR*, volume 2421 of *LNCS*, pages 209–225. Springer, August 2002.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[4] Luca Cardelli and Andrew D. Gordon. Logical properties of name restriction. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 46-60 of *LNCS*, pages 46–60. Springer, May 2001.

[5] Luca Cardelli and Andrew D. Gordon. Ambient logic. Technical report, Microsoft, 2002.

[6] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. Technical Report 2002/01, University of Sussex, 2002.

[7] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[8] Frank Pfenning and Rowan Davies. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, August 2001.

[9] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes (extended abstract). In *IEEE Symposium on Logic in Computer Science*, pages 334–348. IEEE Computer Society Press, June 2000.