Modal Types for Distributed Computation

Jonathan Moody*
jwmoody@cs.cmu.edu

School of Computer Science
Carnegie Mellon University

Abstract. We show that a constructive modal logic with propositions $\Box A$ (necessity) and $\Diamond A$ (possibility) can be interpreted as a type system for a distributed programming language via a Curry-Howard isomorphism. The type $\Box A$ describes mobile terms, whose meaning and well-formedness are location independent, and $\Diamond A$ describes immobile terms which might not have a well-defined meaning outside of a particular location. The programming model assumes no knowledge of concrete locations present in the distributed environment. We give an operational semantics based on processes and an abstract notion of locality. To show how modal types preserve soundness in the presence of *localized* terms, we extend the core calculus with effects and a monadic type of computations $\bigcirc A$. The modal type system ensures that localized values do not escape the location where they are well-defined, and that all effects are executed in some definite location. We also show how to extend the calculus with fixpoints and globally accessible locations, which are needed to express various kinds of recursive computation.

1 A Calculus of Modal Logic

In the model-theoretic presentation of modal logic, truth values for all propositions are determined relative to a "world" with some distinguishing characteristics. In a proof-theoretic development, worlds are treated abstractly, but the consequences of relativized truth remain. Essentially, modal logics have the ability to distinguish various modes of truth for a proposition A, characterizing *where* A is known to be true.

The types, syntax, and static semantics of our calculus are derived from a constructive formalization of modal logic developed by Pfenning and Davies [13]. This was chosen over other intuitionistic formalisms, such as Simpson's [15], since proof reduction and substitution have simple explanations and the logic does not rely on explicit reasoning about worlds and accessibility. The

* This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. The ConCert Project is supported by the National Science Foundation under grant number 0121633: "ITR/SY+SI: Language Technology for Trustless Software Dissemination."

Pfenning/Davies formalism is based on three primitive judgments: A `valid`, meaning that A is true in *every* accessible world; A `true`, meaning that A is locally true "here"; and A `poss`, meaning that A holds in *some* accessible world. Note that validity (A `valid`) is also commonly referred to as necessary truth. These judgments and the propositions $A \to B$ (implication), $\Box A$ (necessity), and $\Diamond A$ (possibility) are defined in relationship to one another, culminating in a natural deduction system for a modal logic obeying axioms characteristic of constructive S4. Logical entailment is given by inference rules for deriving the judgments $\Delta; \Gamma \vdash A$ `true` and $\Delta; \Gamma \vdash A$ `poss`, where Δ are assumptions A `valid`, and Γ are assumptions A `true`. The intuition behind our application of modal logic to distributed programming is the following: If we interpret propositions as types and proofs as programs, it is also quite natural to interpret the logical worlds as sites for computation. Furthermore, we see that validity corresponds to mobility or portability of terms between locations, and possibility corresponds to immobility or locality. A more detailed discussion of the background and logical motivation of the calculus can be found in [11].

For the concrete syntax of our calculus, we adopt the term assignment of Pfenning and Davies [13] with a few extensions. Two sorts of variable (`x` and `u`) are used to represent local hypotheses A `true` and mobile hypotheses A `valid`, respectively. The remote expressions are those objects which are proofs of A `poss`, whereas terms are those which prove A `true`. There is no need for a separate category corresponding to A `valid`, since validity is defined as deduction of A `true` in the absence of locally true assumptions.

$$
\begin{array}{rrcl}
\text{Location Label} & w & ::= & r \mid l \\
\text{Term} & M, N & ::= & r \mid \mathtt{x} \mid \mathtt{u} \mid \lambda \mathtt{x} : A \,.\, M \mid M\ N \\
 & & \mid & \mathtt{box}\, M \mid \mathtt{let\ box\, u = } M \mathtt{\ in\ } N \\
 & & \mid & \mathtt{dia}\, E \\
\text{Remote Expression} & E, F & ::= & l \mid \{M\} \mid \mathtt{let\ box\, u = } M \mathtt{\ in\ } F \\
 & & \mid & \mathtt{let\ dia\, x = } M \mathtt{\ in\ } F
\end{array}
$$

Location labels r and l are used to represent *spatial distribution* of proof terms and expressions during computation, and are not present in the source language. Label r will play a role similar to a hypothesis A `valid`, and l the role of a hypothesis A `poss`.

$$
\begin{array}{rrcl}
\text{Type} & A, B & ::= & A \to B \mid \Box A \mid \Diamond A \\
\text{Runtime Context} & \Lambda & ::= & \cdot \mid \Lambda, r :: A \mid \Lambda, l \div A \\
\text{Mobile Context} & \Delta & ::= & \cdot \mid \Delta, \mathtt{u} :: A \\
\text{Local Context} & \Gamma & ::= & \cdot \mid \Gamma, \mathtt{x} : A \\
\text{Constraint} & \phi, \psi & ::= & \top \mid w \lhd w' \mid w \doteq w' \mid \phi \wedge \psi \\
\text{Location Index} & J & ::= & w \mid J\lhd
\end{array}
$$

Contexts Δ and Γ give types for mobile and local variables, respectively. The runtime context Λ assigns types to labels w, characterizing the distributed en-

vironment in which we regard M or E. The notation $\Lambda\backslash\psi$ is read as "Λ subject to ψ". Constraints ψ will determine which hypotheses in Λ are *accessible* from location J. Finally, J is an index specifying either a particular location ($J = w$), or a kind of quantification over locations accessible from w ($J = w\triangleleft$). We regard $w \triangleleft \triangleleft$ as equivalent to $w\triangleleft$ by definition, so repetitions of $\triangleleft$ are not significant.

Our typing judgments are as follows: $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : A$ is understood to mean that M is a term of type A at location J, under the assumptions $\Lambda\backslash\psi; \Delta; \Gamma$. Similarly, $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J E \div A$ means that expression E has type A at location J. Whenever $J = w\triangleleft$, the judgment $\Lambda\backslash\psi; \Delta; \Gamma \vdash_{w\triangleleft} M : A$ means that M is well-formed at all locations accessible from w.

Only label typing interacts with assumptions $\Lambda\backslash\psi$, hence we abbreviate $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : A$ as $\Delta; \Gamma \vdash_J M : A$ assuming a constant $\Lambda\backslash\psi$ available throughout. The fragment related to $A \to B$ and local computation consists of the usual typing rules for local variables ($\mathtt{x} : A$), lambda abstraction, and application.

$$\frac{}{\Delta; \Gamma, \mathtt{x} : A, \Gamma' \vdash_J \mathtt{x} : A}\ hyp \qquad \frac{\Delta; \Gamma, \mathtt{x} : A \vdash_J M : B}{\Delta; \Gamma \vdash_J \lambda\mathtt{x} : A\,.\,M : A \to B} \to I$$

$$\frac{\Delta; \Gamma \vdash_J M : A \to B \quad \Delta; \Gamma \vdash_J N : A}{\Delta; \Gamma \vdash_J M\ N : B} \to E$$

The necessity ($\Box A$) fragment describes the properties of mobile terms. Operationally speaking, $\mathtt{let\ box\,u = box}\,M\,\mathtt{in}\,N$ will spawn M for evaluation at an arbitrary location. The variable $\mathtt{u} :: A$ gives us access to the value of M in the remainder of the program. Mobility for M is authorized by rule $\Box I$, which requires M be Γ-closed and well-formed at all locations accessible from J.

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

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

The possibility ($\Diamond A$) fragment characterizes computations making use of immobile resources. Conceptually, $\mathtt{let\ dia\,x = dia}\,E\,\mathtt{in}\,F$, sends F to the location where the value of E, bound to ($\mathtt{x} : A$), resides. In cases when $E = l$, F will move to a remote location l, but when $E = \{M\}$ no actual movement of F is required. Mobility of F is authorized by $\Diamond E$, which requires that F be nearly Γ-closed (only the local variable $\mathtt{x} : A$ is permitted), and well-formed at all locations accessible from J. Note that mobile variables ($\mathtt{u} :: A$) in Δ remain available in F, despite the shift in location.

$$\frac{\Delta; \Gamma \vdash_J M : A}{\Delta; \Gamma \vdash_J \{M\} \div A}\ poss \qquad \frac{\Delta; \Gamma \vdash_J M : \Diamond A \quad \Delta; \mathtt{x} : A \vdash_{J\triangleleft} F \div B}{\Delta; \Gamma \vdash_J \mathtt{let\ dia\,x = }M\,\mathtt{in}\,F \div B} \Diamond E$$

$$\frac{\Delta; \Gamma \vdash_J E \div A}{\Delta; \Gamma \vdash_J \mathtt{dia}\,E : \Diamond A} \Diamond I \qquad \frac{\Delta; \Gamma \vdash_J M : \Box A \quad \Delta, \mathtt{u} :: A; \Gamma \vdash_J F \div B}{\Delta; \Gamma \vdash_J \mathtt{let\ box\,u = }M\,\mathtt{in}\,F \div B} \Box E_p$$

The remaining rules characterize location labels r and l, governing which remote locations are available relative to location J. The auxiliary judgment $\psi \vdash^a w \lhd w'$ means that w' is accessible from w under constraints ψ. Constraint entailment $\phi \vdash^a \psi$, presented in section 7.4 of the appendix, defines a small theory of accessibility ($w \lhd w'$) and equivalence ($w \doteq w'$) of locations.

$$\frac{\Lambda = \Lambda_1, r' :: A, \Lambda_2 \quad \psi \vdash^a r' \lhd w}{\Lambda\backslash\psi; \Delta; \Gamma \vdash_w r' : A}\ res \qquad \frac{\Lambda = \Lambda_1, r' :: A, \Lambda_2 \quad \psi \vdash^a r' \lhd w}{\Lambda\backslash\psi; \Delta; \Gamma \vdash_{w\lhd} r' : A}\ ures$$

$$\frac{\Lambda = \Lambda_1, l' \div A, \Lambda_2 \quad \psi \vdash^a w \lhd l'}{\Lambda\backslash\psi; \Delta; \Gamma \vdash_w l' \div A}\ loc$$

The operational intuition behind accessibility and label typing at w is that we may synchronize or pull result values from r' if $r' \lhd w$. Dually, we may jump to a location l' if $w \lhd l'$. In this light, rule *res* and *loc* are obvious. Rule *ures* incorporates an assumption of transitive accessibility, since if $r' : A$ at w then it must also be well-formed at all locations accessible from w. Note that an analogue of *ures* for l' would not be logically sound without symmetric accessibility.

2 Extension with Effects

While the calculus of modal logic suggests that intrinsically immobile things exist, it does not tell us what they are concretely. The terms of the pure calculus are location-neutral, in the sense that all Γ-closed terms M can be boxed as $\texttt{box}\, M : \Box A$ to produce a mobile term. Since the expression language is based on the primitive form of expression $\{M\} \div A$, the encapsulation $(\texttt{dia}\, \{M\}) : \Diamond A$ represents a sort of self-imposed immobility which is not intrinsic to M.

We now instantiate the calculus with a class of objects for which location *inherently* matters. The modal type discipline will assure us that well-formed programs remain safe, despite the presence of location dependent terms. Effectful computations are a suitable example of location dependence for two reasons. First, our choice of where to execute effects may alter the observable behavior of the distributed program. Second, some primitive effects or terms cannot be interpreted correctly when removed from the context of the local machine state.

2.1 Primitive Effects and Typing

We will use a monadic type $\bigcirc A$ to distinguish effectful computations producing A from ordinary pure terms. Other, more precise type systems for effects are possible, but a simple monadic encapsulation of effects is adequate for our purposes. One can motivate the monadic type $\bigcirc A$ through a discussion of lax logic [13], but such a detour is beyond the scope of this paper.

As a simple example, we consider mutable references. References can be integrated into the modal calculus in such a way as to ensure that reference cell values, which are addresses pointing into a local store, never flow between locations.

Secondarily, this preserves structure sharing semantics, and makes synchronized access to shared references easier to implement, since all operations on a cell are performed at *one* definite location.

We introduce a new form of expression, the effectful computation, in addition to the remote expressions of the modal calculus. The reference cell primitives are included directly in the source language as local computations.[1]

$$\begin{array}{rrcl}
\text{Type} & A, B & ::= & \ldots \mid \texttt{unit} \mid \texttt{ref}\, A \\
\\
\text{Term} & M, N & ::= & \ldots \mid \texttt{()} \mid a^w \mid \texttt{comp}\, P \\
\text{Local Computation} & P, Q & ::= & [M] \mid \texttt{let comp x} = M \texttt{ in } Q \\
 & & & \mid \texttt{let box u} = M \texttt{ in } Q \\
 & & & \mid \texttt{ref}\, M \mid \texttt{!}M \mid M \texttt{:=} N \\
\text{Remote Computation} & E, F & ::= & l \mid \{M\} \mid \texttt{let box u} = M \texttt{ in } F \\
 & & & \mid \texttt{let dia x} = M \texttt{ in } F \\
 & & & \mid \{P\} \mid \texttt{let comp x} = M \texttt{ in } F
\end{array}$$

The effectful computations P perform a sequence of primitive effects locally, without jumping to some other location. The remote computations (previously remote expressions) E and F, may now include effects executed here or remotely ($\{P\}$ and $\texttt{let comp x} = M \texttt{ in } F$).

We introduce a new form of judgment $P \rightsquigarrow A$, meaning that P is a local computation of type A. Rule *comp* allows us to regard term M as a trivial computation. Note that rule $\bigcirc I$ for typing suspended computations ($\texttt{comp}\, P$) requires that P be a purely local computation ($P \rightsquigarrow A$). Operationally, the elimination form $\texttt{let comp x} = \texttt{comp}\, P \texttt{ in } Q$ corresponds to sequential evaluation of P followed by Q, binding the result of P to local variable ($\texttt{x} : A$) in Q. Rule $\Box E_l$ plays a role analogous to $\Box E_p$, allowing us to spawn mobile terms for evaluation elsewhere in the course of an effectful computation.

$$\frac{\Delta; \Gamma \vdash_J P \rightsquigarrow A}{\Delta; \Gamma \vdash_J \texttt{comp}\, P : \bigcirc A}\ \bigcirc I \qquad \frac{\Delta; \Gamma \vdash_J M : \bigcirc A \quad \Delta; \Gamma, \texttt{x} : A \vdash_J Q \rightsquigarrow B}{\Delta; \Gamma \vdash_J \texttt{let comp x} = M \texttt{ in } Q \rightsquigarrow B}\ \bigcirc E$$

$$\frac{\Delta; \Gamma \vdash_J M : A}{\Delta; \Gamma \vdash_J [M] \rightsquigarrow A}\ comp \qquad \frac{\Delta; \Gamma \vdash_J M : \Box A \quad \Delta, \texttt{u} :: A; \Gamma \vdash_J Q \rightsquigarrow B}{\Delta; \Gamma \vdash_J \texttt{let box u} = M \texttt{ in } Q \rightsquigarrow B}\ \Box E_l$$

It is especially instructive to compare elimination forms for $\bigcirc$ (above) and $\Diamond$ (reproduced below). Local computations P produce a *local* value, and rule $\bigcirc E$ allows us to assume $\texttt{x} : A$, *in addition* to the others in Γ. On the other hand, the rule for $\Diamond E$ requires us to discard Γ when passing from one location to the remote location where a binding $\texttt{x} : A$ is available. Local term values bound to variables in Γ are *stable* under effects, but not under a jump to some other location.

$$\frac{\Delta; \Gamma \vdash_J M : \Diamond A \quad \Delta; \texttt{x} : A \vdash_{J \triangleleft} F \div B}{\Delta; \Gamma \vdash_J \texttt{let dia x} = M \texttt{ in } F \div B}\ \Diamond E$$

[1] In situations where different locations support different effects, an encoding of primitives as functions $(A_1 * \cdots * A_k) \to \bigcirc B$ could be used. See section 7.1, for example.

Finally, the rules $poss'$ and $\bigcirc E_p$ confer the ability to execute effects remotely. That is, we may mix freely the execution of local effects $\texttt{let comp x} = M \texttt{ in } F$ with jumps to remote locations $\texttt{let dia x} = M \texttt{ in } F$.

$$\frac{\Delta; \Gamma \vdash_J P \leadsto A}{\Delta; \Gamma \vdash_J \{P\} \div A}\ poss' \qquad \frac{\Delta; \Gamma \vdash_J M : \bigcirc A \quad \Delta; \Gamma, \texttt{x} : A \vdash_J F \div B}{\Delta; \Gamma \vdash_J \texttt{let comp x} = M \texttt{ in } F \div B}\ \bigcirc E_p$$

In the context of a store typing Θ mapping addresses a^w to types A, we can describe the typing rules for primitive effects. Θ and $\Lambda \backslash \psi$ are omitted for clarity, except in typing rule *addr*. Θ does not interact with the other typing rules.

$$\text{Store Typing} \quad \Theta_w \ ::= \ \cdot \ \mid \ \Theta_w, a^w : A$$

$$\frac{\Theta = \Theta_1, a^w : A, \Theta_2}{\Theta; \Lambda \backslash \psi; \Delta; \Gamma \vdash_w a^w : \texttt{ref } A}\ addr \qquad \frac{}{\Delta; \Gamma \vdash_J \texttt{()} : \texttt{unit}}\ unit$$

$$\frac{\Delta; \Gamma \vdash_J M : A}{\Delta; \Gamma \vdash_J \texttt{ref } M \leadsto \texttt{ref } A}\ talloc \qquad \frac{\Delta; \Gamma \vdash_J M : \texttt{ref } A}{\Delta; \Gamma \vdash_J \texttt{!} M \leadsto A}\ tget$$

$$\frac{\Delta; \Gamma \vdash_J M : \texttt{ref } A \quad \Delta; \Gamma \vdash_J N : A}{\Delta; \Gamma \vdash_J M \texttt{:=} N \leadsto \texttt{unit}}\ tset$$

The values of type $\texttt{ref } A$ are store addresses a^w. We use superscript w to emphasize the fact that addresses are a form of localized term. The typing rule *addr* only permits us to regard addresses local to w as having type $\texttt{ref } A$ at the definite location $(J = w)$.

In the extended system, there exist coercions between the modalities $\Box$, $\bigcirc$, and $\Diamond$ as follows:

$$\begin{array}{ll} \vdash_J \lambda \texttt{x} : \Box A \,.\, \texttt{let box u} = \texttt{x in u} & \vdash_J \lambda \texttt{x} : A \,.\, \texttt{comp } [x] \\ : \quad \Box A \to A & : \quad A \to \bigcirc A \\ \vdash_J \lambda \texttt{x} : \bigcirc A \,.\, \texttt{dia}\{\texttt{let comp y} = \texttt{x in [y]}\} & \vdash_J \lambda \texttt{x} : A \,.\, \texttt{dia}\{\texttt{x}\} \\ : \quad \bigcirc A \to \Diamond A & : \quad A \to \Diamond A \end{array}$$

So $\Box A$ (a mobile term) is the strongest modality, and $\Diamond A$ (a remote computation) is the weakest. Both A (any local term) and $\bigcirc A$ (a local computation) can be coerced to $\Diamond A$.

3 Operational Semantics

3.1 Preliminary Definitions

The values of the calculus are of three kinds, corresponding to the distinction in typing judgments $V : A$, $V^+ \leadsto A$, and $V^* \div A$.

$$\begin{array}{rrcl} \text{Term Value} & V & ::= & \lambda \texttt{x} : A \,.\, M \ \mid \ \texttt{box } M \ \mid \ \texttt{comp } P \\ & & & \mid \ \texttt{dia } E \ \mid \ \texttt{()} \ \mid \ a^w \\ \text{Comp. Value} & V^+ & ::= & [V] \\ \text{Remote Value} & V^* & ::= & \{V\} \ \mid \ \{V^+\} \end{array}$$

It is convenient in many cases to regard labels r and l as pseudo-values, though they are not proper normal forms. We use the notation $\overline{V}$ to denote a term value or label r. Similarly, we write $\overline{V^*}$, denoting a remote value or label l.

We wish to give an operational interpretation to the calculus which clearly reflects the spatial distribution of program fragments. Location labels w will serve as process identifiers; we will assume no two processes in a configuration share the same label. To explain the semantics of mutable references, a store H is added to each process l. Stores H_l are finite functions mapping addresses a^l to term values $\overline{V}$. Note that the freely mobile terms $\langle r : M \rangle$ do not require a local store.

$$\begin{array}{rrcl} \text{Store} & H_w & ::= & \cdot \mid H_w[a^w \mapsto \overline{V}] \\ \text{Process} & \pi & ::= & \langle r : M \rangle \mid \langle l : H_l \vDash E \rangle \\ \text{Configuration} & C & ::= & \cdot \mid C, \pi \end{array}$$

Configurations associate each r with a term, and each l with a remote expression/computation E. The linear ordering of a process configuration has no special meaning; we will assume process configurations can be rearranged at will.

While one could permit arbitrary recursion among processes through the use of labels, we delay the introduction of such logically unsound, but computationally useful features until section 4. Constraints ψ were introduced in section 1 to describe the allowed dependencies between locations. We say that constraints ψ are *sound* if there are no cycles in accessibility constraints ($\psi \not\vdash^a w \lhd w$). An equivalence constraint $w \doteq w'$ means that w and w' are identical locations. See section 7.4 for definitions and further discussion of constraints.

Note that labeled processes serve as an *abstract* notion of location; we are not committed to any particular scheduling or location binding mechanism assigning processes to host machines. However, such a binding mechanism must respect constraints $w \doteq w'$ governing collocation of processes. Each *distinct* location $\langle l : H \vDash E \rangle$ is assumed to have its own store H. However, the identity of stores $[H]_\psi$ is determined modulo the location equivalence induced by ψ. If $\psi \vdash^a l \doteq l'$ then $\langle l : H \vDash E \rangle$ and $\langle l' : H' \vDash E \rangle$ *share* one store $[H]_\psi = [H']_\psi$.

We can now define the set of well-formed process configurations. The judgment $\psi \vdash^c C : \Lambda$ means that C establishes Λ under constraints ψ. We define an auxiliary store typing judgment $\Lambda \backslash \psi \vdash^s_w H : \Theta$ (store H has type Θ) in the usual way.

$$\begin{array}{rcl} \Lambda \backslash \psi \vdash^s_w H : \Theta & \iff & \mathrm{Dom}(H) = \mathrm{Dom}(\Theta) \\ & & \wedge\ \forall [a^w \mapsto \overline{V}] \in H \ .\ \Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_w \overline{V} : \Theta(a^w) \\ \psi \vdash^c C : \Lambda & \iff & \mathrm{Dom}(C) = \mathrm{Dom}(\Lambda) \\ & & \wedge\ \forall \langle r : M \rangle \in C \ .\ [\,\cdot; \Lambda \backslash \psi; \cdot; \cdot \vdash_{r \lhd} M : \Lambda(r)] \\ & & \wedge\ \forall \langle l : H \vDash E \rangle \in C \ .\ [\Lambda \backslash \psi \vdash^s_l H : \Theta \ \wedge\ \Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l E \div \Lambda(l)] \end{array}$$

The definition of configuration typing requires that every hypothesis in Λ be realized by a process of the correct form, and every process in C has the type

assigned by Λ. Processes are required to be closed with respect to Δ and Γ, but may refer to local store addresses in Θ or labels $r :: A$ or $l \div A$ in Λ subject to accessibility constraints ψ.

3.2 Substitution

We adopt the definitions of substitution from Pfenning and Davies [13] with some modifications to account for location labels r and l and the new syntactic forms arising from the integration of local and remote effectful computations.

Proposition 1 *The following forms of substitution are well-defined: (1)* $[M/\mathtt{x}]$ *applied to N,P,or E. (2)* $[\![M/\mathtt{u}]\!]$ *applied to N,P,or E. (3)* $\langle P/\mathtt{x}\rangle$ *applied to Q or E. (excluding primitive computations P of the forms* $\mathtt{ref}\,M$, $!M$, $M := N$*). (4)* $\langle\!\langle E/\mathtt{x}\rangle\!\rangle$ *applied to F. These forms of substitution obey the properties:*

$$
\begin{array}{llcl}
\Delta;\Gamma,\mathtt{x}:B,\Gamma' \vdash_J N : A & \wedge\ \Delta;\Gamma \vdash_J M : B & \Longrightarrow & \Delta;\Gamma,\Gamma' \vdash_J [M/\mathtt{x}]N : A \\
\Delta;\Gamma,\mathtt{x}:B,\Gamma' \vdash_J F \div A & \wedge\ \Delta;\Gamma \vdash_J M : B & \Longrightarrow & \Delta;\Gamma,\Gamma' \vdash_J [M/\mathtt{x}]F \div A \\
\Delta;\Gamma,\mathtt{x}:B,\Gamma' \vdash_J Q \rightsquigarrow A & \wedge\ \Delta;\Gamma \vdash_J M : B & \Longrightarrow & \Delta;\Gamma,\Gamma' \vdash_J [M/\mathtt{x}]Q \rightsquigarrow A \\
\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash_J N : A & \wedge\ \Delta;\cdot \vdash_{J\lhd} M : B & \Longrightarrow & \Delta,\Delta';\Gamma \vdash_J [\![M/\mathtt{u}]\!]N : A \\
\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash_J F \div A & \wedge\ \Delta;\cdot \vdash_{J\lhd} M : B & \Longrightarrow & \Delta,\Delta';\Gamma \vdash_J [\![M/\mathtt{u}]\!]F \div A \\
\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash_J Q \rightsquigarrow A & \wedge\ \Delta;\cdot \vdash_{J\lhd} M : B & \Longrightarrow & \Delta,\Delta';\Gamma \vdash_J [\![M/\mathtt{u}]\!]Q \rightsquigarrow A \\
\Delta;\Gamma,\mathtt{x}:B \vdash_J Q \rightsquigarrow A & \wedge\ \Delta;\Gamma \vdash_J P \rightsquigarrow B & \Longrightarrow & \Delta;\Gamma \vdash_J \langle P/\mathtt{x}\rangle Q \rightsquigarrow A \\
\Delta;\Gamma,\mathtt{x}:B \vdash_J F \div A & \wedge\ \Delta;\Gamma \vdash_J P \rightsquigarrow B & \Longrightarrow & \Delta;\Gamma \vdash_J \langle P/\mathtt{x}\rangle F \div A \\
\Delta;\mathtt{x}:B \vdash_{J\lhd} F \div A & \wedge\ \Delta;\Gamma \vdash_J E \div B & \Longrightarrow & \Delta;\Gamma \vdash_J \langle\!\langle E/\mathtt{x}\rangle\!\rangle F \div A
\end{array}
$$

Term substitutions $[\![M/\mathtt{x}]\!]$ and $[M/\mathtt{x}]$ are defined in the usual compositional way. Substitutions of computations $\langle P/\mathtt{x}\rangle$ and $\langle\!\langle E/\mathtt{x}\rangle\!\rangle$ require an unusual definition inductive in P or E, the object of substitution.

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

$$
\begin{array}{rcl}
\langle \mathtt{[}M\mathtt{]}/\mathtt{x}\rangle Q & = & [M/\mathtt{x}]Q \\
\langle \mathtt{let\ comp\,y = } M \mathtt{\ in\ } P/\mathtt{x}\rangle Q & = & \mathtt{let\ comp\,y = } M \mathtt{\ in\ } \langle P/\mathtt{x}\rangle Q \\
\langle \mathtt{let\ box\,u = } M \mathtt{\ in\ } P/\mathtt{x}\rangle Q & = & \mathtt{let\ box\,u = } M \mathtt{\ in\ } \langle P/\mathtt{x}\rangle Q
\end{array}
$$

We omit a definition of $\langle P/\mathtt{x}\rangle F$, which is similar to $\langle P/\mathtt{x}\rangle Q$. Omission of clauses for $\langle \mathtt{ref}\ M/\mathtt{x}\rangle$, $\langle !M/\mathtt{x}\rangle$, and $\langle M := N/\mathtt{x}\rangle$ is not problematic, since the operational semantics will reduce primitive effects to $\mathtt{[}\overline{V}\mathtt{]}$ before substitution.

Location labels w of both varieties are insensitive to substitution. The intuition is that labels denote processes containing closed terms or expressions.

$$
\begin{array}{rcl}
[M/\mathtt{x}]w & = & w \qquad [\![M/\mathtt{u}]\!]w = w \\
\langle\!\langle l/\mathtt{x}\rangle\!\rangle F & = & \mathtt{let\ dia\,x = dia}\,l \mathtt{\ in\ } F
\end{array}
$$

This definition of $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F$ is *not* intended to serve as an effective means of reducing $\mathtt{let\ dia\,x = dia}\,l\,\mathtt{in}\,F$ since $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F = \mathtt{let\ dia\,x = dia}\,l\,\mathtt{in}\,F$. Rather, the form $\mathtt{let\ dia\,x = dia}\,l\,\mathtt{in}\,F$ should be regarded as a way to defer or suspend the substitution $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F$ until the expression value denoted by l can be provided. We will provide a special reduction rule (one not based on substitution) specifically for this form of expression.

3.3 Transition Rules

We use the notation of evaluation contexts to represent decomposition of terms into a redex and surrounding context. Term contexts $\mathcal{R}$ are defined so that $\mathcal{R}[\,M\,]$ denotes a term.

$$\text{Term Context}\quad \mathcal{R} ::= [\,] \mid \mathcal{R}\ M \mid \overline{V}\ \mathcal{R} \mid \mathtt{let\ box\,u = }\mathcal{R}\,\mathtt{in}\,N$$

The form of term context $(\overline{V}\ \mathcal{R})$ allows us to postpone synchronization on r in the function position while continuing to reduce in the argument position. Evaluation contexts for expressions are defined so that $\mathcal{S}[\,P\,]$ and $\mathcal{S}[\,M\,]$ denote local computations, and $\mathcal{S}^*[\,M\,]$, $\mathcal{S}^*[\,P\,]$, and $\mathcal{S}^*[\,E\,]$ denote remote computations.

$$\begin{aligned}
\text{Computation Ctxt.}\quad \mathcal{S} ::= {} & [\,] \mid [\mathcal{R}] \mid \mathtt{let\ comp\,x = }\mathcal{R}\,\mathtt{in}\,Q \\
& \mid \mathtt{let\ box\,u = }\mathcal{R}\,\mathtt{in}\,Q \\
& \mid \mathtt{ref}\,\mathcal{R} \mid \mathtt{!}\mathcal{R} \mid \mathcal{R}\,\mathtt{:=}\,N \mid V\,\mathtt{:=}\,\mathcal{R} \\
& \mid \mathtt{let\ comp\,x = comp}\,\mathcal{S}\,\mathtt{in}\,Q \\
\text{Remote Comp. Ctxt.}\quad \mathcal{S}^* ::= {} & [\,] \mid \{\mathcal{R}\} \mid \{\mathcal{S}\} \mid \mathtt{let\ dia\,x = }\mathcal{R}\,\mathtt{in}\,F \\
& \mid \mathtt{let\ box\,u = }\mathcal{R}\,\mathtt{in}\,F \mid \mathtt{let\ comp\,x = }\mathcal{R}\,\mathtt{in}\,F \\
& \mid \mathtt{let\ comp\,x = comp}\,\mathcal{S}\,\mathtt{in}\,F
\end{aligned}$$

A single-step transition in the semantics is stated as $C \setminus \psi \Longrightarrow C' \setminus \psi'$ for constraints ψ, ψ' and process configurations C, C'. We take the point of view that constraints ψ are informative assertions about the structure of the running program. As additional processes are created, the set of constraints ψ will grow, but we are required to preserve soundness (acyclicity) of ψ and well-formedness of C with respect to ψ ($\psi \vdash^c C : \Lambda$).

Term reduction rules occur in two forms, one applicable to terms $\mathcal{R}[\,M\,]$, the other for $\mathcal{S}^*[\,M\,]$. We follow a convention of naming the variants *app* and *app**, respectively. Processes irrelevant to a reduction step are elided. The rule *app** is straightforward. In a process l, When we encounter a redex $(\lambda\mathtt{x} : A\,.\,M_1')\ \overline{V_2}$ where $\overline{V_2}$ is a term pseudo-value (a value or the label r), we perform substitution of $\overline{V_2}$ for the local variable $\mathtt{x}$ in the function body $([\overline{V_2}/\mathtt{x}]M_1')$. The reduction step is purely local: no terms move from one process to another, and constraints ψ are unchanged. The variant form *app* is similar, though it occurs in a context $\mathcal{R}[\,]$.

$$\frac{V_1 = \lambda\mathtt{x} : A\,.\,M_1'}{\langle l : H \vDash \mathcal{S}^*[\,V_1\ \overline{V_2}\,]\rangle \setminus \psi \Longrightarrow \langle l : H \vDash \mathcal{S}^*[\,[\overline{V_2}/\mathtt{x}]M_1'\,]\rangle \setminus \psi}\ \mathit{app}^*$$

The *letbox** rule and variants (*letbox*, *letbox*$_l$, *letbox*$_p$)[2] govern the evaluation of mobile boxed terms of type $\Box A$. When we reach a redex of the form $\texttt{let box u = box}\, M\, \texttt{in} \ldots$, an independent process is spawned for evaluation of M at a fresh location r'. Movement of M from l to r' is justified by the typing rule $\Box I$, since well-formed M is closed and cannot refer to labels l' or local store addresses a^l. The result label r' is substituted for $\texttt{u}$ in N. Label r' will serve as a placeholder for the value of M, allowing us to achieve some concurrency in evaluation.

$$\frac{\begin{array}{l} V = \texttt{box}\, M \quad r' \texttt{ fresh} \\ \psi' \;=\; \psi \;\wedge\; (r' \lhd l) \;\wedge\; (\bigwedge_i \{r_i \lhd r' \mid \psi \vdash^a r_i \lhd l\}) \end{array}}{\begin{array}{c} \langle l : H \vDash \mathcal{S}^*[\,\texttt{let box u} = V\, \texttt{in}\, N\,]\rangle \setminus \psi \\ \Longrightarrow \langle r' : M\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,[\![r'/\texttt{u}]\!]N\,]\rangle \setminus \psi' \end{array}}\ \textit{letbox}^*$$

Note that when $\langle r' : M\rangle$ is created, we add certain constraints to ψ characterizing its relationship to other processes. The original process l becomes dependent on r' ($r' \lhd l$). And since term M may depend on other mobile terms r_i, we assert $r_i \lhd r'$ for all r_i such that $r_i \lhd l$.

Synchronization on a result label r' may happen nondeterministically, but becomes necessary when the structure of a value is observed. In rule *syncr**, notice that the process $\langle r' : \overline{V}\rangle$ has no local store, hence $\overline{V}$ must be a pure, location-neutral term which may be moved safely to l.

$$\frac{}{\langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,\overline{V}\,]\rangle \setminus \psi}\ \textit{syncr}^*$$

The combination of *letbox**, *syncr**, and the treatment of r as a lazy pseudo-value is reminiscent of the `future` and `touch` mechanisms of Multilisp [10]. But in a statically typed framework we can guarantee such concurrency is harmless, since the spawned $\langle r' : M\rangle$ cannot execute effects and has no access to a shared store.

For the fragment of the calculus relating to effects, we have a general rule for sequential evaluation of computations, as well as some effect-specific primitives. We omit the variant rule *seq*$_p$ for reducing $\texttt{let comp x} = M\, \texttt{in}\, F$.

$$\frac{}{\langle l : H \vDash \mathcal{S}^*[\,\texttt{let comp x} = \texttt{comp}\,[\overline{V}]\,\texttt{in}\, Q\,]\rangle \setminus \psi \Longrightarrow \langle l : H \vDash \mathcal{S}^*[\,\langle [\overline{V}]/\texttt{x}\rangle Q\,]\rangle \setminus \psi}\ \textit{seq}$$

Effects are never executed in a context $\mathcal{R}[\,]$, only in contexts $\mathcal{S}^*[\,]$. The definitions of contexts $\mathcal{S}$ and $\mathcal{S}^*$ allow us to reduce $\texttt{let comp x} = \texttt{comp}\, P\, \texttt{in}\,(\ldots)$ to $\texttt{let comp x} = \texttt{comp}\,[\overline{V}]\,\texttt{in}\,(\ldots)$. So rules *seq* and *seq*$_p$ are operationally adequate, given the following reduction rules for primitive effects. Now $H(a^l)$ denotes lookup of the value associated with a^l, and $H[a^l \mapsto \overline{V}]$ denotes extending

[2] *letbox* is identical to *letbox** except it applies in a context $\mathcal{R}[\,]$. *letbox*$_l$ and *letbox*$_p$ apply to redices $\texttt{let box u} = \texttt{box}\, M\, \texttt{in}\, Q$ and $\texttt{let box u} = \texttt{box}\, M\, \texttt{in}\, F$ respectively.

or updating the store H with a binding $[a^l \mapsto \overline{V}]$.

$$\frac{a^l \texttt{ fresh} \quad H' = H[a^l \mapsto \overline{V}]}{\langle l : H \vDash \mathcal{S}^*[\,\texttt{ref}\,\overline{V}\,]\rangle \setminus \psi \Longrightarrow \langle l : H' \vDash \mathcal{S}^*[\,[a^l]\,]\rangle \setminus \psi}\ alloc$$

$$\frac{H(a^l) = \overline{V}}{\langle l : H \vDash \mathcal{S}^*[\,!a^l\,]\rangle \setminus \psi \Longrightarrow \langle l : H \vDash \mathcal{S}^*[\,[\overline{V}]\,]\rangle \setminus \psi}\ get$$

$$\frac{H' = H[a^l \mapsto \overline{V}]}{\langle l : H \vDash \mathcal{S}^*[\,a^l := \overline{V}\,]\rangle \setminus \psi \Longrightarrow \langle l : H' \vDash \mathcal{S}^*[\,[()]\,]\rangle \setminus \psi}\ set$$

All reduction rules for effects are local and involve no communication. But recall that stores H are identified modulo $\psi \vdash^a l \doteq l'$, so updates to H will affect all processes at the same location implicitly.

Finally, the *letdia* and *syncl* rules define the behavior of terms of type $\Diamond A$. For redex $\texttt{let dia x = dia}\,E\,\texttt{in}\,F$, we simply substitute the computation E for $\texttt{x}$ in F using expression substitution $\langle\!\langle E/\texttt{x}\rangle\!\rangle F$. This operation rearranges the structure of the computation locally; no actual movement between locations occurs. The restriction $E \neq l'$ is crucial because substitution of a label $\langle\!\langle l'/\texttt{x}\rangle\!\rangle F$ does not allow us to make progress.

$$\frac{V = \texttt{dia}\,E \quad E \neq l'}{\langle l : \texttt{let dia x} = V\,\texttt{in}\,F\rangle \setminus \psi \Longrightarrow \langle l : \langle\!\langle E/\texttt{x}\rangle\!\rangle F\rangle \setminus \psi}\ letdia$$

One can look at *syncl* as a sort of dual of *syncr* — instead of bringing the immobile expression E to our current location, the mobile computation F is sent to the location of E, a pseudo-value. Mobility of F is justified by the typing rule $\Diamond E$ since a well-formed F is closed (with the exception of $\texttt{x} : A$) and cannot refer to labels l' or local store addresses a^l.

$$\frac{V = \texttt{dia}\,l' \quad l'' \texttt{ fresh} \quad \psi' \;=\; \psi \;\wedge\; (l' \doteq l'')}{\begin{array}{l}\langle l : H \vDash \texttt{let dia x} = V\,\texttt{in}\,F\rangle,\ \langle l' : H' \vDash \overline{V*}\rangle \setminus \psi \\ \Longrightarrow \langle l : H \vDash l''\rangle,\ \langle l' : H' \vDash \overline{V*}\rangle,\ \langle l'' : H' \vDash \langle\!\langle \overline{V*}/\texttt{x}\rangle\!\rangle F\rangle \setminus \psi'\end{array}}\ syncl$$

Duplication of $\overline{V*}$ from process $\langle l' : H' \vDash \overline{V*}\rangle$ as $\langle l'' : H' \vDash \langle\!\langle \overline{V*}/\texttt{x}\rangle\!\rangle F\rangle$ is needed to assure type preservation in cases when more than one process might jump to l'. We add the assertion $l' \doteq l''$ to ψ indicating that l' and l'' share the *same* location and store H'. So the creation of process l'' and duplication of $\overline{V*}$ is purely local, not requiring any movement.

3.4 Properties

Theorem 2 (Type Preservation) *If ψ is sound (accessibility is acyclic), process configuration C is well-formed ($\psi \vdash C : \Lambda$), and a reduction step $C \setminus \psi \Longrightarrow C' \setminus \psi'$ is made, then ψ' remains sound and $\psi' \vdash^c C' : \Lambda'$, where Λ' extends Λ.*

Proof: By cases on the $C \setminus \psi \Longrightarrow C' \setminus \psi'$ judgment. A few cases crucial to safety and preservation of locality (*syncr'*, *letbox'*, and *syncl*) are presented in the appendix, section 7.2. See also [11].

Theorem 3 (Progress) *Assume ψ is sound (accessibility is acyclic). If $\psi \vdash^c C : \Lambda$, then either C is terminal (all processes contain values) or $C \setminus \psi \Longrightarrow C' \setminus \psi'$.*

Proof: Consider an arbitrary process $\langle r : M \rangle$ or $\langle l : H \vDash E \rangle$ in C. We reformulate the progress theorem as follows, separating M or E from the rest of the configuration C.

$$\begin{array}{ll} & \psi \ \mathtt{sound} \ \wedge \ \psi \vdash^c C : \Lambda \ \wedge \ \cdot;\Lambda\backslash\psi;\cdot;\cdot \vdash_J M : A \quad (\text{where } J = r\triangleleft) \\ \Longrightarrow & M = V \ \vee \ \exists C', M' \,.\, C, \langle r : M \rangle \setminus \psi \Longrightarrow C', \langle r : M' \rangle \setminus \psi' \\ & \psi \ \mathtt{sound} \ \wedge \ \psi \vdash^c C : \Lambda \ \wedge \ \Lambda\backslash\psi \vdash^s_l H : \Theta \\ & \wedge \ \Theta;\Lambda\backslash\psi;\cdot;\cdot \vdash_J E \div A \quad (\text{where } J = l \text{ or } J = l\triangleleft) \\ \Longrightarrow & E = V^* \ \vee \ \exists C', E' \,.\, C, \langle l : H \vDash E \rangle \setminus \psi \Longrightarrow C', \langle l : H' \vDash E' \rangle \setminus \psi' \end{array}$$

The proof then proceeds by induction on the order of location indices J imposed by accessibility constraints ψ, with nested induction on the structure of typing derivations for M and E. Indices J are compared by their root labels w ignoring quantifier symbols. We first prove the property for judgments of the form $J\triangleleft$, in which case our induction hypothesis is that progress holds for *prior* J' ($J' \triangleleft J$). Then unquantified J can be considered under the hypothesis that progress holds for *subsequent* J' ($J \triangleleft J'$). For details of a proof for the core calculus see [11]. The same strategy extends to the fragment with effects.

4 Extension with Recursion

4.1 Fixpoint Constructs

We consider two natural forms of fixpoint corresponding to the distinction between variables $(\mathtt{u} :: A)$ and $(\mathtt{x} : A)$. We refer to $\mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M$ as a valid or mobile fixpoint, and $\mathtt{fix}\,(\mathtt{x} : A)\,.\,M$ as local fixpoint. The operational semantics is given in the conventional way, with substitution used to perform unrolling. See section 4 of appendix.

$$\frac{\Delta, \mathtt{u} :: A; \cdot \vdash_{J\triangleleft} M : A}{\Delta; \Gamma \vdash_J \mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M : A}\ fix_v \qquad \frac{\Delta; \Gamma, \mathtt{x} : A \vdash_J M : A}{\Lambda\backslash\psi; \Delta; \Gamma \vdash_J \mathtt{fix}\,(\mathtt{x} : A)\,.\,M : A}\ fix$$

The treatment of expression fixpoint over computations P or remote computations E is less obvious. For reasons of conceptual economy and uniformity, we adopt an approach of encoding such fixpoints with fix_v or fix.

For example, $\mathtt{fixv}\,(\mathtt{u} :: \Diamond A)\,.\,\mathtt{dia}\,E$ binds a fixpoint variable $(\mathtt{u} :: \Diamond A)$ in E. In the body E, the idiom $\mathtt{let\ dia\,x = u\ in}\,F$ represents a recursive jump to an unrolled copy of E. When and if E terminates without making such a nested jump, we continue with F. Note that the expression E must be Γ-closed, a consequence of using $(\mathtt{u} :: \Diamond A)$ rather than a local fixpoint variable $(\mathtt{x} : \Diamond A)$.

It seems clear that mobile fixpoint over $(\mathtt{u} :: \Diamond A)$ is a useful idiom. But local fixpoints $\mathtt{fix}\,(\mathtt{x} : \Diamond A)\,.\,\mathtt{dia}\,E$ are not, since the scope of $(\mathtt{x} : \Diamond A)$ is so limited. On the other hand, $\mathtt{fix}\,(\mathtt{x} : \bigcirc A)\,.\,\mathtt{comp}\,P$ does seems useful for expressing recursion over a purely local computation P.

4.2 Globally Accessible Locations

Fixpoints of remote computations should allow us to jump repeatedly between distinct locations, perhaps executing some local effects at each. Such nontrivial forms of $\mathtt{fixv}\,(\mathtt{u} :: \Diamond A)\,.\,\mathtt{dia}\,E$ require a set of assumptions $(\mathtt{v_i} :: \Diamond A_i)$ in Δ, representing the locations amongst which a program can jump. But how can such mobile assumptions of type $\Diamond A_i$ be realized? Essentially, the difficulty is that we can conclude $\Lambda \backslash \psi \vdash_w \mathtt{dia}\,l : \Diamond A$, but *not* $\Lambda \backslash \psi \vdash_{w \lhd} \mathtt{dia}\,l : \Diamond A$. To make the latter conclusion sound, we must know that l is accessible from *any* other location.

In prior development, we imposed a condition that accessibility $(\psi \vdash^a w \lhd w')$ be acyclic so that recursion among processes could not arise. We now make an exception to this condition for a class of *globally accessible* locations. We introduce new forms of accessibility constraint as follows. These formulae have the obvious meanings under constraint entailment.

$$\text{Constraint} \quad \phi, \psi \ ::= \ \ldots \ \mid \ \forall \mathtt{w}\,.\,\mathtt{w} \lhd l \ \mid \ \forall \mathtt{w}\,.\,r \lhd \mathtt{w}$$

Accessibility may now be cyclic, permitting recursion among processes. But it is intended that this feature be used judiciously to represent the initial distributed environment in which a program runs. For example, such locations l could hold bindings for global resources $(\mathtt{v_i} :: \Diamond A_i)$. Programmers cannot create cyclic configurations, since none of the rules of the operational semantics introduce this form of constraint. We also extend typing with a rule *uloc* for labels l, allowing us to conclude $\vdash_{w \lhd} l' \div A$. For details see the appendix, section 7.5.

5 Related Work

The most closely related work is the λ_{rpc} calculus developed independently and concurrently by Jia and Walker [9]. The type system of λ_{rpc} is inspired by a spatial interpretation of modal logic, though it is an extension of S5 (not S4) with some hybrid-logic features. They show that the semantics is type sound in the presence of reference cells (store addresses). But to ensure that locality of addresses is preserved, Jia and Walker do not permit evaluation of e under $\mathtt{close}(\lambda \mathtt{p}\,.\,e)$ (their form of $\Box$ introduction). We can allow concurrent evaluation under $\mathtt{box}\,M$ because the monadic type $\bigcirc A$ and judgment $P \rightsquigarrow A$ isolate effectful computations from pure terms.

The hybrid-logic aspect of the λ_{rpc} type system, which introduces explicit worlds (absolute locations) and edge names (relative paths between locations), makes comparison to a standard modal logic difficult. Also, we believe the introduction of explicit worlds and names into the calculus has consequences for

portability. λ_{rpc} reveals the network topology and permits one to implement more efficient algorithms specialized to that topology, but such programs are less portable.

The ambient logic and ambient calculus are also related to our work. Though the ambient calculus itself was not logically motivated, Cardelli and Gordon [6, 5] and Caires and Cardelli [1, 2] have developed an ambient logic with modal operators to characterize the behavior of ambient calculus programs. In their work, accessibility is interpreted as containment of ambients, $\Box\Psi$ requires all sub-locations satisfy Ψ, and $\Diamond\Psi$ requires the existence of some sub-location satisfying Ψ. Some typical relevant properties are: how the shape of the ambient configuration changes over time or whether the scope of an ambient name escapes another ambient. As with names in the Pi-calculus, untyped ambients have no fixed locality or scope; in the absence of a specification, nested ambients may move freely in and out of other ambients.

Cardelli, Ghelli, and Gordon have also developed a static type system for ambients [3, 4] which restricts ambient mobility. Ambients can be declared immobile relative to others via name restriction $(\nu n : Amb^{Y\,Z'}[^{Z}T])\,P$, where type decorations Z' and Z control objective and subjective movements of the ambient n. Since the authors view mobility as a declared behavioral property extrinsic to the ambient names themselves, it is natural to allow "immobile" ambients to move when contained inside mobile ones, for example. This differs somewhat from our notions of mobility and immobility which were derived from logical necessity and possibility. Mobility of terms in our calculus is naturally an inherited property, in that mobile terms may only depend on other mobile terms.

An advantage (or limitation) of the constructive approach we demonstrate in this paper is that all well-formed programs automatically obey a certain safety policy, preserving locality of certain term values and resources encapsulated as $\Diamond A$. This safety is intrinsic in the criteria for program well-formedness. The ambient logic approach is a flexible theoretical tool for characterizing program behavior, but lacks some good properties of a type system, such as decidability, and a manageable level of complexity that programmers can grasp.

Finally, there are systems derived from the Pi-calculus and ML which have no intrinsic notion of locality. Typically, the Pi-calculus imposes no restrictions on the scope (mobility) of names, but some researchers have pursued type systems which give names a static scope tied to a particular location. Names communicated outside of this natural scope are tracked by the type system. See work on DPI by Hennessey *et al.* [8] and $lsd\pi$ by Ravara, Matos, *et al.* [14]. In his PhD thesis [12] Moreira explores a type inference framework for a distributed ML which analyzes the locality of references. Rather than ruling out mobility of references in all cases, Moreira's goal is to eliminate proxies and synchronization when it can be established that a reference is local.

6 Conclusions

We have presented a calculus for distributed computation based on constructive S4 modal logic. The modal propositions $\Box A$ and $\Diamond A$ can quite naturally be regarded as types describing mobile and immobile elements of the language. We gave a concrete example of what these immobile elements might be by extending the calculus with effects and a monadic type $\bigcirc A$ characterizing effectful computations. Effects are a natural example to use, since machine state is quite difficult to move or replicate at runtime. The modal type discipline provides a safe way to mix mobility with effects and localized terms in a distributed computation. The spatial modalities $\Box$ and $\Diamond$ interact with $\bigcirc$ to determine where effects are executed and restrict the mobility of localized terms.

Applications of this work to ConCert [7], a grid-programming environment based on code certification via safety proofs, are particularly important to us. The ConCert system assumes a peer-to-peer overlay network with unreliable participants, so long-running stateful computations are to be discouraged. We speculate that the absence of axiom $\Diamond A \to \Box\Diamond A$ (characteristic of S5) is beneficial in such a setting. It corresponds to creation of a mobile proxy for a value of any type A; managing these proxies and handling failures could become a burden on the runtime system. Furthermore, modal logics and calculi based on an explicit-worlds formalism lead to a more explicit programming model and strong assumptions about the network topology. But in an ad-hoc peer-to-peer network, these assumptions are not helpful to the programmer; the overlay may emulate the assumed topology, but the emulation will likely bear no resemblance to the performance characteristics of the physical network. Such assumptions might, however, play a useful role in limiting communication or other resources used by a program.

Researchers are meeting with success in applying modal logic to distributed computation, but just as there are many modal logics, there are now a growing number of distributed calculi. The goal of capturing (im)mobility properties is shared among researchers, and can be achieved with a modal type system. But some open questions remain. The choice to base the calculus directly on proof terms of a logic, and S4 modal logic in particular, affects the programming model a great deal. Though we have developed a few examples, not much is known about the practicality of programming in this setting as opposed to a calculus derived from another logic, or one of the process calculus formalisms (typed mobile ambients, DPI, *etc.*). Besides these issues of usability and expressive power, choice of one formalism over another may require different implementation strategies, or raise issues of feasibility which we have not yet explored. But in any case, it seems clear that modal logic is a powerful conceptual framework in which we may think about the design of distributed programming languages.

Acknowledgments Thanks go to Frank Pfenning for many discussions and guidance on this topic, and to Joshua Dunfield for comments on a draft of this paper.

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, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languagese and Programming, 26th International Colloquium (ICALP)*, volume 1644 of *LNCS*, pages 230–239. Springer, 1999.
4. Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. Technical Report MSR-TR-99-32, Microsoft, June 1999.
5. 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.
6. Luca Cardelli and Andrew D. Gordon. Ambient logic. Technical report, Microsoft, 2002.
7. Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. In *GRID 2002 Workshop*.
8. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
9. Limin Jia and David Walker. Modal proofs as distributed programs. July 2003.
10. D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90. ACM Press, 1989.
11. Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, October 2003.
12. Álvaro Moreira. *A Type-Based Locality Analysis for a Functional Distributed Language*. PhD thesis, University of Edinburgh, 1999.
13. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, August 2001.
14. António Ravara, Ana G. Matos, Vasco T. Vasconcelos, and Luís Lopes. Lexically scoped distribution: what you see is what you get. In *Foundations of Global Computing*. Elsevier, 2003.
15. Alex K. Simpson. *Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

7 Appendix

7.1 Examples

A *marshalling function* $A \to \Box A$ can be implemented for any *observable* type A. However, such a function may be very large and/or inefficient. Some primitive marshalling functions on integers, floating point, and string values can be provided without changing the logical character of the system. They preserve type safety since the structure of most simple term values does not permit any dependency on other values or local machine state. The following example shows how to lift a primitive function `marshall_int::int -> □ int` to operate on lists of integers.

```
let box (marshal_int_list::int list -> □ (int list)) =
    box
        fix marshall . λ lst .
            case lst of
                nil => box nil
              | cons(x,tl) =>
                    let box vx = marshall_int x in
                    let box vtl = marshall tl in
                        box cons(vx,vtl)
```

In cases such as this, when the boxed term is already a value, it would be highly desirable to inline the operation `let box u = box` V `in` $(\ldots)$. By this we mean simply performing the substitution $[\![V/\mathtt{u}]\!]$ without generating an intermediate process. This is consistent with the intuition that $\Box A$ captures mobility – the value V may move, but is not *forced* to move to some remote location. Since a^w is abstract, in the sense that there is no language mechanism to observe its internal structure, a programmer cannot construct `ref` $A \to \Box($`ref` $A)$. Nor would we want to provide a primitive coercion `ref` $A \to \Box$`ref` A to make reference cells portable.

Marshalling functions establish (logically) the mobility of a value but perform no useful work. Using $\Box A$, we can also spawn non-value terms for concurrent evaluation at an arbitrary location. We consider the example of a distributed implementation of the Fibonacci function.

```
let (fib:□int -> int) =
    fixv f . λ bn .
        let box n = bn in
        if n < 2 then
            n
        else
            let box f1 = box f (box (n-1)) in
            let box f2 = box f (box (n-2)) in
                f1 + f2
```

Note that in this case, we must use the mobile fixpoint `fixv`, since the function itself must be mobile. The code `let box u = box` M `in` ... is an idiom for spawning M for parallel evaluation, similar to `(let (u (future` M`)) ...)` in Multilisp [10].

In the possibility ($\Diamond A$) fragment, recall there is no actual movement without primitive remote resources $\Diamond A$. In this example, each such remote resource provides a set of effect primitives encapsulated as functions $A \to \bigcirc B$. The effects involved are I/O operations interacting with a network printer and the home console. Let the environment be characterized by Δ_0:

```
server :: ◇{submit : doc -> ○job, wait : job -> ○string}

home :: ◇{read_doc : string -> ○doc, write : string -> ○unit}
```

Variable `server` represents a place where two primitive effects are available: `submit` and `wait`. Variable `home` represents a location where we can `read_doc` (read a document from a file) or `write` messages to the console. Given bindings for these mobile variables, and marshalling functions `marshall_string : string -> □string` and `marshal_doc : doc -> □doc`, we can write the following program which prints a document remotely.

```
let dia h_env = home in

let (remote_print:doc -> ◇ unit) =
    λ x .
        dia
        let box p = marshal_doc x in
        let dia s_env = server in
        let comp j = s_env.submit p in
        let comp s = s_env.wait j in
        let box sv = marshal_string s in
        let dia h_env = home in
        let comp _ = h_env.write sv in
            {()}
     in

let comp d = val (h_env.read_doc ``filename'') in
let dia _ = remote_print d in
    {()}
```

Note that the use of $\Diamond$ and/or $\bigcirc$ imposes a sequential style of programming. The function `remote_print` executes a sequence of effects (`let comp`) and jumps (`let dia`) causing the document `d` to be printed remotely and a status message written on the `home` console. Marshalling functions `marshal_doc` and `marshal_string` are used to make the document and the status message portable between locations. Also note that `j : job`, a local handle used to refer to print jobs, disappears from scope when we jump to `home`. If type `job` is held abstract, the value of `j` cannot be removed from the location `server`.

7.2 Type Preservation (selected cases)

Case:

$$\frac{}{\langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,\overline{V}\,]\rangle \setminus \psi}\ \mathit{syncr}^*$$

$\Lambda \backslash \psi \vdash^s_l H : \Theta$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathcal{S}^*[\,r'\,] \div A$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l r' : B$	Typing Inv. Lemma
$\cdot; \Lambda \backslash \psi; \cdot; \cdot \vdash_{r' \lhd} \overline{V} : B$	Assumption, Definition
$\psi \vdash^a r' \lhd l$	Inversion (res)
$\cdot; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \overline{V} : B$	Natural Mobility
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \overline{V} : B$	Weakening
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathcal{S}^*[\,\overline{V}\,] \div A$	Ev. Context Typing
$\psi' = \psi$ and ψ' `sound`	Assumption
$\Lambda' = \Lambda$	Directly

Case:

$$\frac{\begin{array}{l} V = \mathtt{box}\, M \quad r' \ \mathtt{fresh} \\ \psi' \ = \ \psi \ \wedge \ (r' \lhd l) \ \wedge \ (\bigwedge_i \{r_i \lhd r' \mid \psi \vdash^a r_i \lhd l\}) \end{array}}{\begin{array}{c} \langle l : H \vDash \mathcal{S}^*[\,\mathtt{let\ box\,u = } V \,\mathtt{\ in\ } N\,]\rangle \setminus \psi \\ \Longrightarrow \langle r' : M\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,[\![r'/\mathtt{u}]\!]N\,]\rangle \setminus \psi' \end{array}}\ \mathit{letbox}^*$$

$\Lambda \backslash \psi \vdash^s_l H : \Theta$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathcal{S}^*[\,\mathtt{let\ box\,u = } V \,\mathtt{\ in\ } N\,] \div C$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathtt{let\ box\,u = } V \,\mathtt{\ in\ } N : B$	Typing Inv. Lemma
$\Theta; \Lambda \backslash \psi; \mathtt{u} :: A; \cdot \vdash_l N : B$	Inversion ($\Box E$)
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathtt{box}\, M : \Box A$	Assumption, Inversion ($\Box E$)
$\cdot; \Lambda \backslash \psi; \cdot; \cdot \vdash_{l \lhd} M : A$	Inversion ($\Box I$)
Let $\Lambda' = \Lambda, r' :: A$	
$\psi' = \psi \wedge (r' \lhd l) \wedge (\bigwedge_i \{r_i \lhd r' \mid \psi \vdash_w r_i \lhd l\})$	Assumption
$\psi \vdash^a \phi \implies \psi' \vdash^a \phi$	Entailment $\vdash^a$
$\psi' \vdash^a r_i \lhd l \implies \psi' \vdash^a r_i \lhd r'$	Entailment $\vdash^a$
$\psi' \vdash^a r' \lhd l$	Entailment $\vdash^a$
$\cdot; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_{r' \lhd} M : A$	Mobility Against Accessibility
$\Theta; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_{l \lhd} r' : A$	Typing ($ures$)
$\Theta; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_l [\![r'/\mathtt{u}]\!]N : B$	Weakening, Substitution
$\Theta; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_l \mathcal{S}^*[\,[\![r'/\mathtt{u}]\!]N\,] \div C$	Weakening, Ev. Context Typing
$\Lambda' \backslash \psi' \vdash^s_l H : \Theta$	Weakening

r' `fresh`	Assumption
$\exists w, w' \,.\, \psi' \vdash^a w \lhd w'$ contradicts ψ `sound`	Entailment $\vdash^a$
ψ' `sound`	By Contradiction
$\Lambda' \supseteq \Lambda$	Directly

Case:

$$\frac{V = \mathtt{dia}\, l' \quad l'' \ \mathtt{fresh} \quad \psi' \ = \ \psi \ \wedge \ (l' \doteq l'')}{\begin{array}{l}\langle l : H \vDash \mathtt{let\ dia\,x = }V\ \mathtt{in}\, F\rangle,\ \langle l' : H' \vDash \overline{V*}\rangle \setminus \psi \\ \Longrightarrow \langle l : H \vDash l''\rangle,\ \langle l' : H' \vDash \overline{V*}\rangle,\ \langle l'' : H' \vDash \langle\!\langle \overline{V*}/\mathtt{x}\rangle\!\rangle F\rangle \setminus \psi'\end{array}}\ \mathit{syncl}$$

$\Lambda \backslash \psi \vdash^s_l H : \Theta$	Assumption, Definition
$\Lambda \backslash \psi \vdash^s_{l'} H' : \Theta'$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathtt{let\ dia\,x = }V\ \mathtt{in}\, F \div B$	Assumption, Definition
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_{l'} \overline{V*} \div A$	Assumption, Definition
$\cdot; \Lambda \backslash \psi; \cdot; \mathtt{x} : A \vdash_{l \lhd} F \div B$	Inversion ($\Diamond E$)
$\Theta; \Lambda \backslash \psi; \cdot; \cdot \vdash_l \mathtt{dia}\, l' : \Diamond A$	Assumption, Inversion ($\Diamond E$)
$\psi \vdash^a l \lhd l'$	Inversion (loc)
Let $\Lambda' = \Lambda, l'' \div B$	
$\psi' = \psi \wedge (l' \doteq l'')$	Assumption
$\psi \vdash^a \phi \implies \psi' \vdash^a \phi$	Entailment $\vdash^a$
$\psi' \vdash^a l' \doteq l''$	Entailment $\vdash^a$
$\psi' \vdash^a l \lhd l''$	Entailment $\vdash^a$ (cong)
$\Theta'; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_{l''} \overline{V*} \div A$	Weakening, Eq. Worlds ($l' \doteq l''$)
$\Theta'; \Lambda' \backslash \psi'; \cdot; \mathtt{x} : A \vdash_{l'' \lhd} F \div B$	Weakening, Natural Mobility
$\Theta'; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_{l''} \langle\!\langle \overline{V*}/\mathtt{x}\rangle\!\rangle F \div B$	Substitution
$\Theta; \Lambda' \backslash \psi'; \cdot; \cdot \vdash_l l'' \div B$	Typing (loc)
$\Lambda' \backslash \psi' \vdash^s_{l''} H' : \Theta'$	Weakening, Eq. Worlds ($l' \doteq l''$)
$l''\ \mathtt{fresh}$	Assumption
$\exists w, w' \ .\ \psi' \vdash^a w \lhd w'$ contradicts ψ `sound`	Form of ψ', Entailment $\vdash^a$
ψ' `sound`	By Contradiction
$\Lambda' \supseteq \Lambda$	Directly

7.3 Fixpoint Semantics

The typing rules and operational semantics of the term fixpoints are as follows:

$$\frac{\Delta, \mathtt{u} :: A; \cdot \vdash_{J \lhd} M : A}{\Delta; \Gamma \vdash_J \mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M : A}\ \mathit{fix}_v \qquad \frac{\Delta; \Gamma, \mathtt{x} : A \vdash_J M : A}{\Delta; \Gamma \vdash_J \mathtt{fix}\,(\mathtt{x} : A)\,.\,M : A}\ \mathit{fix}$$

$$\frac{}{\begin{array}{l}\langle l : H \vDash \mathcal{S}^*[\,\mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M\,]\rangle \setminus \psi \\ \Longrightarrow \langle l : H \vDash \mathcal{S}^*[\,[\![\mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]\!]M\,]\rangle \setminus \psi\end{array}}\ \mathit{unroll}^*_v$$

$$\frac{}{\begin{array}{l}\langle l : H \vDash \mathcal{S}^*[\,\mathtt{fix}\,(\mathtt{x} : A)\,.\,M\,]\rangle \setminus \psi \\ \Longrightarrow \langle l : H \vDash \mathcal{S}^*[\,[\mathtt{fix}\,(\mathtt{x} : A)\,.\,M/\mathtt{x}]M\,]\rangle \setminus \psi\end{array}}\ \mathit{unroll}^*$$

Of course, variant rules unroll_v and unroll exist for reduction of fixpoints in a term context $\mathcal{R}[\ ]$. Note that substitution $[\![\mathtt{fixv}\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]\!]M$ is type sound because we stipulate $\Delta, \mathtt{u} :: A; \cdot \vdash_{J \lhd} M : A$ in the typing rule fix_v.

Though it might seem that $\texttt{fixv}\,(\texttt{u} :: A)\,.\,M$ could be encoded as a local fixpoint $\texttt{fix}\,(\texttt{x} : \Box A)\,.\,\texttt{let box u = x in}\,(\texttt{box}\,M)$, this encoding does not have the desired operational behavior. Regardless of the form of M, such a representation unrolls forever without termination.

7.4 Theory of Locations

Accessibility and equivalence of locations determines the permissible dependencies between processes in a configuration C. Recall that w denotes a location (process label) r or l. We will think about such labels as abstract locations or worlds in a Kripke semantics of modal logic.

$$\text{Constraint } \phi, \psi \;::=\; \top \;\mid\; w \lhd w' \;\mid\; w \doteq w' \;\mid\; \phi \wedge \psi$$

A primitive constraint $(w \lhd w')$ asserts that accessibility holds between w and w'. The constraint $w \doteq w'$ asserts the equivalence of w and w' under accessibility. That is, both have the same accessibility properties with respect to all other worlds, so in a sense they represent (or share) the same location. Compound constraints are conjunctions of such primitive constraints, or the unit element $\top$. When convenient, we may regard a formula ϕ as a set of primitive constraints, joined implicitly by conjunction.

Equivalence $(w \doteq w')$ obeys reflexivity, symmetry, and transitivity, but does *not* entail $w \lhd w'$ or $w' \lhd w$ directly. Our notion of accessibility $w \lhd w'$ obeys transitivity (from S4) and respects congruence classes of worlds (as defined by $\doteq$). The S4 assumption of reflexivity $(w \lhd w)$ is not made explicit in the theory of locations, but is present in the term and expression typing rules. Constraints ψ govern the accessibility of *remote* terms (r) and expressions (l); appeals to reflexive accessibility are made via typing rules *hyp** $(\vdash_J \texttt{u} : A)$ and *poss* $(\vdash_J \{M\} \div A)$. In fact, including a reflexivity axiom $\psi \vdash^a w \lhd w$ would have the undesirable effect of allowing recursive processes such as $\langle r : r \rangle$.

The judgment $\Gamma \vdash^a \psi$, capturing entailment for constraints, is defined as follows. In this context, Γ denotes a set of constraints $\phi_1, \phi_2, \ldots, \phi_n$.

$$\frac{}{\Gamma, \psi \vdash^a \psi} \qquad \frac{\Gamma, \phi_1, \phi_2 \vdash^a \psi}{\Gamma, (\phi_1 \wedge \phi_2) \vdash^a \psi}$$

$$\frac{}{\Gamma \vdash^a w \doteq w} \qquad \frac{\Gamma \vdash^a w \doteq w'}{\Gamma \vdash^a w' \doteq w} \qquad \frac{\Gamma \vdash^a w \doteq w' \quad \Gamma \vdash^a w' \doteq w''}{\Gamma \vdash^a w \doteq w''}$$

$$\frac{\Gamma \vdash^a w \doteq w_1 \quad \Gamma \vdash^a w_1 \lhd w_2 \quad \Gamma \vdash^a w_2 \doteq w'}{\Gamma \vdash^a w \lhd w'} \qquad \frac{\Gamma \vdash^a w \lhd w' \quad \Gamma \vdash^a w' \lhd w''}{\Gamma \vdash^a w \lhd w''}$$

The specification above is only intended to be complete for derivation of primitive conclusions $w \lhd w'$ or $w \doteq w'$, not an arbitrary formula ψ.

7.5 Extension to Globally Accessible Locations

Globally accessible locations l are permitted if we modify the theory of locations determined by constraint entailment $\Gamma \vdash^a \psi$. We introduce a new form of constraint, $\forall \mathtt{w}\,.\,\mathtt{w} \lhd l$ with the intuitive meaning that l is accessible from anywhere. We also add the dual constraint for labels r, those from which every other location is accessible.

$$\text{Constraint } \phi, \psi ::= \ \ldots \quad | \quad \forall \mathtt{w}\,.\,\mathtt{w} \lhd l \quad | \quad \forall \mathtt{w}\,.\,r \lhd \mathtt{w}$$

$$\frac{\Gamma \vdash^a \forall \mathtt{w}\,.\,\mathtt{w} \lhd l}{\Gamma \vdash^a w \lhd l}\ [w] \qquad \frac{\Gamma \vdash^a \forall \mathtt{w}\,.\,r \lhd \mathtt{w}}{\Gamma \vdash^a r \lhd w}\ [w]$$

The two inference rules are parametric in w allowing us to instantiate the quantifier with any world w. Note that there is no introduction form for $\forall \mathtt{w}\,.\,\mathtt{w} \lhd l$. This is by design; the constraint $\forall \mathtt{w}\,.\,\mathtt{w} \lhd l$ is a primitive assertion about l that must be introduced explicitly.

Given the new form of constraint $\forall \mathtt{w}\,.\,\mathtt{w} \lhd l$, we can now express a new typing rule for labels l.

$$\frac{\Lambda = \Lambda_1, l' \div A, \Lambda_2 \quad \psi \vdash^a \forall \mathtt{w}\,.\,\mathtt{w} \lhd l'}{\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w \lhd} l' \div A}\ \mathit{uloc}$$

The rule permits typing of l in the context of a mobile term or expression, where such occurrences were not typeable before. For example, we may now conclude $\Lambda \backslash \psi; \Delta; \cdot \vdash_{w \lhd} \mathtt{dia}\, l : \Diamond A$, which allows us to realize assumptions $\mathtt{u} :: \Diamond A$ in Δ.