Thesis Proposal:
Type Theory for Mobility and Locality

Jonathan Moody
jwmoody@cs.cmu.edu

February 13, 2004

Abstract

In this thesis, we consider distributed programming from a type-theoretic, logical perspective. We develop a calculus for distributed programming based on a constructive modal logic. Through a Curry-Howard interpretation of proof terms as programs and propositions as types, we show that logical necessity is connected to mobility, and possibility to remote locality. We give an operational interpretation based on process configurations, where each process serves as an abstract location. We then show how to extend the calculus in various ways, raising the question of what a proper type-theoretic notion of immobility would be. Finally, we discuss issues that arise when one considers implementation of the calculus, both promising opportunities and practical difficulties.

1 Summary

The enterprise of designing programming languages for distributed computation raises issues of locality and concurrency in the execution of programs — locality, since fragments of code and data comprising the program may be spatially dispersed, and concurrency, since it is natural to assume spatially distributed machines or logically distributed processes are not synchronized or coupled during execution. Historically, distributed languages have differed greatly in the means by which disparate fragments of a program may interact, and to what extent a programmer should be aware of the means of interaction, its semantics, and its limitations.

From a systems-building perspective, questions of locality, mobility, and immobility can be settled based on implementation technology. That is, the mobile entities are simply those for which the language runtime implements marshalling, and locality is a question of efficiency, not a semantically relevant property. Following this approach, it is quite possible to make bad design decisions — there may be a variety of irregular conditions or restrictions on mobility, or perhaps no way to state or enforce a fixed mobility policy. Everyone recognizes that some values are more difficult to move between locations than others. But there seems to be no broad agreement on *why* some values are marshallable and others not, or what constitutes a *correct* implementation of marshalling.

In this thesis, we reconsider distributed programming from a type-theoretic, logical perspective. We develop a calculus for distributed programming based on a constructive modal logic. Through a Curry-Howard interpretation of proof terms as programs and propositions as types, we show that logical necessity is connected to mobility, and possibility to remote locality. We give an operational interpretation based on process configurations, where each process serves as an abstract location.

We then show how to extend the calculus with datatypes and effectful computation, and discuss general criteria for attributing mobility or immobility to terms under each new extension. By considering carefully the typing rules (logical content) of each extension, we can, in many cases, distinguish mobile and immobile values in a principled way. We also consider the programming model induced by the calculus, which includes idioms for remote evaluation at an arbitrary location and access to a remote resource at a particular location.

The plan for future work includes a consideration of polymorphism and abstract types. This should further clarify the distinction between mobile and immobile values in the calculus. We speculate that abstract types $\exists\alpha\,.\,A$ and the hidden nature of the implementation type α can explain immobility in a clean, type-theoretic way. Towards an implementation, we discuss resource discovery and binding, scheduling processes to be run on concrete host machines, implementation of synchronization, and distributed garbage collection.

1.1 Background

Distributed programming is not a new concept, people have been building languages or language libraries for distributed computation for about 20 years. See Bal *et al.* [1] for a comprehensive (though dated) survey of distributed languages. The authors view concurrency, communication, and handling of failures as the distinguishing features of a distributed language, and categorize each language by how it provides such facilities. Theoretical models of distribution and mobility lagged behind, with the π-calculus [15] formalism being introduced slightly over a decade ago. Many variants of the π-calculus have followed, based on alterations in the form and semantics of communication. There seems to be a lot of room for experimentation in this area, and no particular distributed programming paradigm or theoretical formalism has achieved universal acceptance.

Thus language designers have taken many approaches in packaging distributed computation for the programmer. For example, communication is obviously required for implementation of a distributed language; without it parts of a program at different locations cannot interact. But communication may or may not be revealed explicitly to the programmer. Logically shared memory and remote procedure calls, hide communication behind an abstraction, while message passing or read/write channels reveal communication explicitly.

In the design of some languages, the ideal is to provide location transparent distributed execution, either through marshalling the values as necessary, or through liberal use of proxies. Implementations often make tradeoffs between runtime cost, implementation complexity, and the ideal of a location transparent semantics. The essential difficulty with this approach is that each extension of the language with new types and values may break location-transparency in some way. Marshalling some kinds of non-portable values will trigger a runtime error or may invoke an alternative mechanism such as marshalling by proxy. Because such a language usually has no static notion of mobility or location, marshalling errors are not detectable at compile time, nor can the compiler easily distinguish proxies from local values for purposes of optimization. We note, however, that it is sometimes possible to reconstruct partial locality information through an analysis/elaboration step [17].

Alternatively, one can add facilities for code mobility and distributed communication to a language, giving these primitives the desired operational semantics. Agent or process mobility and/or value communication over channels then allows interaction among remote locations. This becomes a general backdoor through which all sorts of protocols or mobility policies can be implemented. Given the operational semantics, one can then design a type system to limit mobility to the cases where it makes sense, or the decision can be deferred to the programmer. Since communication is explicit, one may assume the programmer is aware that the receiver might be remote and is also aware of the consequences of passing values across the channel. In these situations, it is up to the programmer to ensure that the communicated value is interpreted properly at the remote location.

1.1.1 The Process Calculi

The formal process calculi are based on a model of communication and passing of values on channels. If channel names themselves can be passed (requiring a higher-order channel), this can be interpreted as a kind of mobility, since processes can communicate with a variety of other processes over their lifetime. This highly abstract notion of "location" leads to a kind of unrestricted mobility, since the scope of channel names cannot be easily determined. A structural scope-extrusion rule for names allows changing their scope as needed to account for their actual occurrences.

A variety of alternative calculi and type systems have been proposed to reflect location more concretely and characterize the mobility of processes, values, or names. The ambient calculus, for example, reflects location using the ambient notation $n[P]$ representing a process P situated in the ambient (location) $n[\,]$. Indeed, ambients *replace* named channels as a means of communication. As with names in the π-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.

Though the ambient calculus itself was not logically motivated, Cardelli and Gordon [8, 7] and Caires and Cardelli [3, 4] 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 to 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.

Cardelli, Ghelli, and Gordon have also developed a static type system for ambients [5, 6] 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.

The safe ambients formalism of Levi and Sangiorgi [14] is an attempt to rectify some of the unforseen problems with the operational semantics of ambients. Their criticism of the ambient calculus centers on "grave interferences", in which the non-determinism of reduction can create various unexpected outcomes and stuck states. The solution is to add co-capabilities to the ambient calculus, such that both parties in a primitive interaction agree. This makes it possible to analyze the behavior of programs in a more modular fashion, since capabilities and co-capabilities fit together in a way analogous to introduction and elimination forms for types.

There are other systems derived from the π-calculus which impose a static notion of locality for channel names. See work on DPI by Hennessey *et al.* [9] and $lsd\pi$ by Ravara, Matos, *et al.* [19]. These systems assign a kind of existential type, reminiscent of our type

$\Diamond A$, to names which are communicated outside of their natural, static scope. Processes may only use channel names of this type by jumping into the scope (location) where that channel is defined.

1.1.2 Logically Motivated Calculi

Very closely related to our work is the λ_{rpc} calculus developed independently and concurrently by Jia and Walker [11]. The type system of λ_{rpc} is also inspired by a spatial interpretation of modal logic, though it is an extension of S5 (not S4) with some hybrid-logic features. 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, it would seem that the use of explicit worlds and edge names leads to a fundamentally different programming model, in which the programmer orchestrates the distribution of program fragments at runtime.

Borghuis [2] has also made a connection between modal logic and distributed systems. But in Borghuis' work, the interpretation of the modalities is quite different from ours. He uses type $\Box_w(A \to B)$ to represent a service running at location w which implements the function $A \to B$. The modality $\Box_w$ is a non-standard modality which has no connection to our interpretation of $\Box$. However, Borghuis' $\Box_w$ serves to distinguish remote resources, and therefore resembles $\Diamond$ in our calculus. He does not discuss the semantics of the logic in terms of accessibility, but seems to assume that any location is accessible from any other.

1.1.3 Real Systems

Java Remote Method Invocation (RMI) and object serialization is a typical example of the integration of distributed computation and mobility into a programming language. As much as possible, Java RMI hides the distinction between local and remote objects. Method calls on remote objects are handled by marshalling parameters and result values with a generic object serialization facility. Subtyping on classes and interfaces essentially hides the distinction between local and remote objects. Both the actual remote object and proxy implementations share a common interface, by casting the proxy object to that interface, a programmer can ignore the distinction in most situations. Because indirection is inherent in Java method calls, distinguishing proxies from ordinary local objects at runtime imposes no additional performance cost.

Marshalling (called serialization) is handled as follows: Certain primitive values and all objects derived from classes implementing the `Serializable` interface are marshallable by conversion to a stream of bytes; this stream representation allows the objects to be transmitted and reconstituted elsewhere. The Java Serialization API enforces a runtime requirement that all fields of a serializable object contain serializable objects recursively. Such a constraint cannot be enforced statically since it is not expressible as a subclass relationship in the Java type system. Users are allowed to override the default object serialization methods, so it is hard to make broad statements about the properties of serialization as a whole.

2 Type Theory and Logical Motivation

In this proposal, we adopt a type-theoretic approach to distributed programming and mobility. We will show how to interpret constructive modal logic as a calculus for distributed computation and modal propositions as types characterizing mobility and locality of terms. As a result, computation in the calculus is value-oriented, resembling the family of λ-calculi more than the communication-oriented process-calculi. Mobility arises out of a natural operational interpretation of introduction and elimination for $\Box A$ and $\Diamond A$, not as a side-effect of introducing message passing or communication over channels. At the same time, this work differs from attempts to give a completely location-transparent account of distributed computation. Though communication is not explicit, the modal type system distinguishes mobility or locality properties of terms. This allows us to examine mobility $\Box A$ and locality $\Diamond A$ in the same logical, type-theoretic framework as other extensions of the calculus, such as algebraic datatypes and effectful computation.

2.1 A Calculus of Modal Logic

In the model-theoretic presentation of modal logic, the truth value of a proposition is determined relative to a world — each world having perhaps 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 [18]. This was chosen over other intuitionistic formalisms, such as Simpson's [20], since proof reduction and substitution have simple explanations and the logic does not rely on explicit reasoning about worlds and accessibility. The Pfenning/Davies formalism is based on three primitive judgments on A, a proposition: A `true`, meaning that A is locally true "here"; A `valid`, meaning that A `true` holds in *every* accessible world; and A `poss`, meaning that A `true` holds in *some* accessible world. 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 $\Delta; \Gamma \vdash A\ \texttt{true}$ and $\Delta; \Gamma \vdash A\ \texttt{poss}$, 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 locality. A more detailed discussion of the background and logical motivation of the calculus can be found in [16].

Moving to a framework with explicit proofs, judgments A `true` and A `poss` become the analytic judgments $M : A$ (M proves A `true`) and $E \div A$ (E proves A `poss`), where M and E are proof objects. For the concrete syntax of proofs, we adopt the term assignment of Pfenning and Davies [18]. Δ and Γ become variable typing contexts; two sorts of variable (`x` and `u`) are used to represent local hypotheses A `true` and mobile hypotheses A `valid`, respectively.

$$
\begin{array}{rccl}
\text{Proposition (Type)} & A, B & ::= & A \to B \quad | \quad \Box A \quad | \quad \Diamond A \\
\text{Valid (Mobile) Context} & \Delta & ::= & \cdot \quad | \quad \Delta, \mathtt{u} :: A \\
\text{True (Local) Context} & \Gamma & ::= & \cdot \quad | \quad \Gamma, \mathtt{x} : A
\end{array}
$$

The 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.

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

We now explain the essential logical content of the deduction rules for proofs of truth $\Delta; \Gamma \vdash M : A$ and possibility $\Delta; \Gamma \vdash E \div A$. Of course these can also be viewed as term and expression typing judgments for a programming language.

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

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

The fragment pertaining to the connective $\to$ is not unusual, following the usual definition of implication (function typing). In the introduction rule $\to I$ we may conclude $A \to B$ if, assuming A `true` we can prove B `true`. In the elimination rule $\to E$, we can conclude B `true` given A `true` and $A \to B$ `true`. The hypothesis rule *hyp* is similarly straightforward. One should note that $\to I$ introduces a locally true hypothesis $\mathtt{x} : A \in \Gamma$, not a mobile, valid hypothesis.

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

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

The rules pertaining to $\Box A$ (necessarily A) are understood as follows. The introduction form $\Box I$ allows us to internalize A `valid` as $\Box A$ `true`, noting that the proof term M proves A in absence of locally true assumptions. The elimination form allows us to introduce a new valid hypothesis $\mathtt{u} :: A$ given $\Box A$ `true`, and reasoning with this new hypothesis prove B `true`. The (valid) hypothesis rule *hyp** allows us to conclude A `true` given A `valid`. The soundness of this rule relies on reflexivity of accessibility — A `valid` is intended to mean A is true at every world accessible from "this" world, so A `true` at "this" world. This completes the definition of the judgment $\Delta; \Gamma \vdash M : A$.

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

Truth and possibility are related by the rules *poss* and $\Box E_p$. Rule *poss* states that A `true` at "this" world means A `poss`. As with *hyp**, this incorporates reflexivity of accessibility since "this" world is an accessible world. Rule $\Box E_p$ corresponds to $\Box E$, but allows introduction of a hypothesis $\mathtt{u} :: A$ justified by $\Box A$ `true` in the course of a deduction of B `poss`.

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

The connective $\Diamond A$ (possibly A) has an introduction rule $\Diamond I$ in which we internalize A `poss` as $\Diamond A$ `true`. Intuitively, if A is true somewhere, then $\Diamond A$ is true at "this" world. In the elimination rule, given $\Diamond A$ `true` and a proof B `poss` under the assumptions $\Delta; \mathtt{x} : A$, we may conclude B `poss`. Note that the structure of proof $F \div B$ must be *independent* of any locally true assumptions, with the sole exception of $\mathtt{x} : A$. Intuitively, if we know A `true` at some accessible world, and $\Delta; \mathtt{x} : A \vdash F \div B$, then B `poss` since F relies on no other assumptions and may in some sense be sent to the world where A `true`.

3 Operational Interpretation

3.1 Representing Spatial Distribution

While it is possible to define natural syntactic reductions directly on proof terms M and expressions E, this is somewhat unsatisfying and not revealing of the spatial content of modal types. Our goal in this section is to show that the worlds of modal logic can indeed be interpreted as sites for computation. When interpreted as a typing judgment, $\mathtt{box}\, M : \Box A$ will mean that M has type A in *all* accessible locations, and hence M denotes a mobile term of type A. Similarly, $\mathtt{dia}\, E : \Diamond A$ will that E produces (or denotes directly) a term of type A at *some* accessible location.

We wish to give an operational interpretation to the calculus which clearly reflects the spatial distribution of program fragments. Hence processes are introduced to serve as abstract locations in which terms and expressions reside. Location labels w will serve as process identifiers; we will assume no two processes in a configuration share the same label.

$$\begin{array}{rrcl}
\text{Location Label} & w & ::= & r \mid l \\
\text{Process} & \pi & ::= & \langle r : M \rangle \mid \langle l : E \rangle \\
\text{Configuration} & C & ::= & \cdot \mid C, \pi
\end{array}$$

The two varieties of location label allow us to distinguish between term and expression processes. Configurations associate each r with a term $\langle r : M \rangle$, and each l with an expression $\langle l : E \rangle$. The linear ordering of a process configuration has no special meaning; we will assume process configurations can be rearranged at will.

$$\begin{array}{rrcl}
\text{Term} & M, N & ::= & r \mid \mathtt{x} \mid \mathtt{u} \mid \ldots \\
\text{Remote Expression} & E, F & ::= & l \mid \{M\} \mid \ldots
\end{array}$$

The language of terms is extended with labels r, and expressions with labels l. Intuitively, a label w occurring in a proof term refers to some remote term or expression in process w. So process configurations consist of a mutually-referential collection of labeled terms or expressions. We give a precise specification of well-formedness for process configurations in the appendix, section 11.1, which excludes cyclic dependencies among processes. Acyclicity of the dependency graph allows us to prove a progress theorem for the semantics, and implies soundness and completeness with respect to the pure logical proof system [16].

3.2 Form of Values

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

$$\begin{array}{rrcl}
\text{Term Value} & V & ::= & \lambda \mathtt{x} : A \,.\, M \mid \mathtt{box}\, M \mid \mathtt{dia}\, E \\
\text{Remote Exp. Value} & V^* & ::= & \{V\}
\end{array}$$

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 an remote value or label l.

3.3 Substitution

We adopt the definitions of substitution from Pfenning and Davies [18] with some modifications to account for location labels r and l. Term substitutions $[\![M/\mathtt{x}]\!]$ and $[M/\mathtt{x}]$ are defined in the usual compositional way. Substitutions of expressions $\langle\!\langle E/\mathtt{x}\rangle\!\rangle$ require an unusual definition inductive in E, the object of substitution.

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

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

$$\begin{array}{rclcrcl}
[M/\mathtt{x}]w & = & w & \qquad & [\![M/\mathtt{u}]\!]w & = & w \\
\langle\!\langle l/\mathtt{x}\rangle\!\rangle F & = & \multicolumn{5}{l}{\mathtt{let\ dia}\,\mathtt{x} = \mathtt{dia}\,l\,\mathtt{in}\,F}
\end{array}$$

This definition of $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F$ is *not* intended to serve as an effective means of reducing $\mathtt{let\ dia}\,\mathtt{x} = \mathtt{dia}\,l\,\mathtt{in}\,F$ since $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F = \mathtt{let\ dia}\,\mathtt{x} = \mathtt{dia}\,l\,\mathtt{in}\,F$. Rather, the form $\mathtt{let\ dia}\,\mathtt{x} = \mathtt{dia}\,l\,\mathtt{in}\,F$ 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.

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

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

Proof: By a straightforward induction. See [16]. □

3.4 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. Evaluation contexts for expressions are defined so that $\mathcal{S}^*[\,M\,]$ denotes a remote expression.

$$\begin{array}{rcll}
\text{Term Context} & \mathcal{R} & ::= & [\,] \mid \mathcal{R}\ M \mid \overline{V}\ \mathcal{R} \mid \mathtt{let\ box}\,\mathtt{u} = \mathcal{R}\,\mathtt{in}\,N \\
\text{Remote Exp. Ctxt.} & \mathcal{S}^* & ::= & [\,] \mid \{\mathcal{R}\} \mid \mathtt{let\ dia}\,\mathtt{x} = \mathcal{R}\,\mathtt{in}\,F \\
 & & & \mid \mathtt{let\ box}\,\mathtt{u} = \mathcal{R}\,\mathtt{in}\,F
\end{array}$$

The form of term context $(\overline{V}\ \mathcal{R})$ will allow us to postpone synchronization on r in the function position while continuing to reduce in the argument position. Expression redices are accommodated in the empty context $\mathcal{S}^* = [\,]$

A single-step transition in the semantics is stated as $C \Longrightarrow C'$ for process configurations C, C'. The semantics encodes concurrency as non-deterministic choice among processes in C, and does not address issues of scheduling or location binding for assigning processes to host machines. Processes irrelevant to a reduction step are elided.

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. The rules *app* and *app** are straightforward. To reduce $(\lambda \mathtt{x} : A\,.\,M_1')\ \overline{V_2}$, 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.

$$\frac{V_1 = \lambda \mathtt{x} : A\,.\,M_1'}{\langle r : \mathcal{R}[\,V_1\ \overline{V_2}\,]\rangle \Longrightarrow \langle r : \mathcal{R}[\,[\overline{V_2}/\mathtt{x}]M_1'\,]\rangle}\ \mathit{app}$$

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

The *letbox* rule and variants (*letbox**, *letbox*$_p$) govern the evaluation of mobile boxed terms of type $\Box A$. When we encounter a redex of the form ($\mathtt{let\ box\,u=box}\,M\,\mathtt{in}\,\ldots$), an independent process is spawned for evaluation of M at a fresh location r'. Term M is known to be arbitrarily mobile by the typing rule $\Box I$ which requires that M be closed with respect to local variables Γ. The fresh label r' is substituted for $\mathtt{u}$ in N. Label r' will serve as a placeholder for the value of M, allowing us to achieve some concurrency in evaluation.

$$\frac{V = \mathtt{box}\,M \quad r'\ \mathtt{fresh}}{\begin{array}{ll} & \langle r : \mathcal{R}[\,\mathtt{let\ box\,u=}V\,\mathtt{in}\,N\,]\rangle \\ \Longrightarrow & \langle r' : M\rangle,\ \langle r : \mathcal{R}[\,[\![r'/\mathtt{u}]\!]N\,]\rangle \end{array}}\ \mathit{letbox}$$

We have omitted *letbox** and *letbox*$_p$, which are trivial variations of the rule above.

Synchronization on a result label r' may happen nondeterministically, but becomes necessary when the structure of a value is observed. We omit the variant rule *syncr**. The safe mobility of value $\overline{V}$ is assured because it was produced by evaluation of a closed, mobile term. Logically speaking, this is a natural consequence of type-preserving reduction, but special care must be taken when we integrate effects into the language.

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

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 [12]. But in a pure language we know such concurrency is harmless, since the spawned process $\langle r' : M\rangle$ cannot access a shared store or execute effects.

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

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

Finally, 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 (nearly) closed with respect to local variables Γ, the exception being $\mathtt{x} : A$ which will be provided at the destination.

$$\frac{V = \mathtt{dia}\, l' \quad l'' \doteq l'}{\begin{array}{rl} & \langle l : \mathtt{let\ dia\, x = } V \,\mathtt{in}\, F\rangle,\ \langle l' : \overline{V^*}\rangle \\ \Longrightarrow & \langle l : l''\rangle,\ \langle l' : \overline{V^*}\rangle,\ \langle l'' : \langle\!\langle \overline{V^*}/\mathtt{x}\rangle\!\rangle F\rangle \end{array}} \; syncl$$

Duplication of $\overline{V^*}$ from process $\langle l' : \overline{V^*}\rangle$ as $\langle l'' : \langle\!\langle \overline{V^*}/\mathtt{x}\rangle\!\rangle F\rangle$ is needed to assure type preservation in cases when more than one process might jump to l'. We choose to interpret the creation of l'' as a sort of aliasing, with process l'' sharing the *same* concrete location as l'. This is made more precise in the discussion of constraint formulae $l'' \doteq l'$ in the appendix, section 11.3.

3.5 Properties

When well-formedness for process configurations is suitably defined, the operational semantics satisfies type preservation, progress, strong normalization, and confluence (modulo a notion of equivalence accounting for lazy synchronization).

However, to properly state these theorems we need much more machinery for characterizing the well-formed process configurations. For progress and strong normalization we must restrict our attention to process configurations with an acyclic dependency structure. To track dependencies we introduce accessibility constraints, and the operational semantics is modified to propagate and update the set of constraints. See section 11.1 and [16] for details.

4 Induced Programming Model

The calculus can be decomposed into several nested sub-languages organized around combinations of the type constructors $\rightarrow$, $\Box$ and $\Diamond$. Each of these fragments determines a programming model, and there is a natural progression in expressivity from the sub-language containing only $\rightarrow$ (functions and application) to the full complement of $\rightarrow$, $\Box$, and $\Diamond$ type constructors and their introduction and elimination forms.

4.1 $\rightarrow$ Fragment

At the core, we have a λ-calculus fragment consisting of abstraction, application, and ordinary, local variables $\mathtt{x} : A$. If we restrict ourselves to just these constructs, we of course, recover the programming model of the λ-calculus and our operational semantics is purely local. When a well-formed program M is placed in process r, reduction proceeds locally $\langle r : M\rangle \Longrightarrow^* \langle r : V\rangle$, without spawning or interaction with other processes.

4.2 $\rightarrow, \Box$ Fragment

Now we proceed to consider the λ-calculus fragment ($\rightarrow$) in conjunction with the necessity fragment $\mathtt{box}\, M$ and $\mathtt{let\ box\, u =} M \,\mathtt{in}\, N$. By introducing type $\Box A$, we have a way to characterize those terms which are potentially mobile. The typing rule for $\Box I$ (logical necessitation) expresses the condition that $\mathtt{box}\, M$ is mobile if M is Γ-closed. There is no particular location where M must be evaluated, since it has been established that $\mathtt{box}\, M : \Box A$. Hence we interpret the box elimination forms $\Box E$ as spawning a new process at an arbitrary location for evaluation of M.

In the combined $\rightarrow, \Box$ fragment we have a programming model of the λ-calculus with a concurrent, remote evaluation mechanism. In this fragment, a well-formed program M, when placed in a process r, behaves as $\langle r : M\rangle \Longrightarrow^* C, \langle r : V\rangle$. Evaluation of M proceeds concurrently with all processes spawned (directly and indirectly) by M.

4.3 $\rightarrow, \Box, \Diamond$ Fragment

Finally, when the possibility fragment, $\mathtt{dia}\, E$, $\mathtt{let\ dia\, x} = M\, \mathtt{in}\, F$, etc. is added, we see the potential for self-directed mobility of programs. Introducing type $\Diamond A$ allows us to refer to remote resources of type A, and the elimination form allows jumping to that location. However, without some primitive initial assumptions about the distributed environment, all such values $\mathtt{dia}\, E : \Diamond A$ refer to trivially remote resources. If we admit $\mathtt{u_i} :: \Diamond A_i$ bound to values $\mathtt{dia}\, l_i$, then programs may use these values as capabilities to jump amongst the locations denoted by l_i. Evaluation of a program E in the full calculus takes the form $C_0, \langle l : E\rangle \Longrightarrow^* C_0, C, \langle l : V^*\rangle$, where C_0 consists of $\langle l_i : V_i^*\rangle$ representing a set of available remote resources.[1]

4.4 Role of Accessibility

The notion of accessibility, imported from the realm of the semantics of modal logic, plays a key role in determining the properties of a modal logic. Indeed, logicians have investigated a variety of modal logics differing only in the assumed properties of this accessibility relation. The system K is characterized by reflexivity, S4 by reflexivity and transitivity, and S5 by the inclusion of symmetry in addition to reflexivity and transitivity.

Obviously, accessibility plays a role in the static semantics of the calculus, since it was derived from a constructive modal logic (S4). But what effect does each of these assumptions have on the programming model for a distributed calculus? Reflexivity permits trivial "local" uses of the $\Box A$ and $\Diamond A$ constructs of the language. For example, a value of type $\Diamond A$ does not necessarily encapsulate a remote resource, but a true remote resource must be encapsulated as $\Diamond A$ in a local computation. Transitivity implies an abstraction of "distance" between locations — the precise number of hops between here and an accessible location is not particularly relevant (consider the axiom schemas $\Box A \rightarrow \Box\Box A$ and $\Diamond\Diamond A \rightarrow \Diamond A$). Though not present in the calculus presented here, we conjecture that symmetry, by eliminating the "directionality" of accessibility between locations, admits marshalling of proxies. But in the absence of symmetry, we are required to adopt a value-copying interpretation of marshalling, in which terms are moved in their totality.

5 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 $\mathtt{box}\, M : \Box A$ to produce a mobile term. Since the expression language is based on the primitive form of expression $\{M\} \div A$, the encapsulation $(\mathtt{dia}\,\{M\}) : \Diamond A$ 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,

[1] There is actually a slight complication; we need a primitive notion of *globally accessible* locations l_i which are accessible from all others. This is only troublesome in that it introduces cycles and thus potential non-termination. The calculus itself retains the character of S4 and we do not generally assume symmetric accessibility. See section 6.2.

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.

5.1 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 [18], 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 primitive operations on reference cells are included directly in the source language as local computations.[2]

$$
\begin{array}{rrcl}
\text{Type} & A, B & ::= & \ldots \mid \bigcirc A \mid \texttt{ref}\, A \mid \texttt{1} \\[1ex]
\text{Term} & M, N & ::= & \ldots \mid \texttt{comp}\, P \mid \texttt{()} \\
\text{Local Computation} & P, Q & ::= & \texttt{[}M\texttt{]} \mid \texttt{let comp}\,\texttt{x} = M\, \texttt{in}\, Q \\
& & & \mid \texttt{let box}\,\texttt{u} = M\, \texttt{in}\, Q \\
& & & \mid \texttt{ref}\, M \mid \texttt{!}M \mid M \texttt{:=} N \\
\text{Remote Computation} & E, F & ::= & l \mid \{M\} \mid \texttt{let box}\,\texttt{u} = M\, \texttt{in}\, F \\
& & & \mid \texttt{let dia}\,\texttt{x} = M\, \texttt{in}\, F \\
& & & \mid \{P\} \mid \texttt{let comp}\,\texttt{x} = M\, \texttt{in}\, F
\end{array}
$$

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 $\texttt{let comp}\,\texttt{x} = M\,\texttt{in}\,F$).

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 $\texttt{let comp}\,\texttt{x} = \texttt{comp}\,P\,\texttt{in}\,Q$ 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.

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

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

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

[2]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 8, for example.

$\mathtt{x} : A$, *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 $\mathtt{x} : A$ is available. Local term values bound to variables in Γ are *stable* under effects, but not under a jump to some other location.

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

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 $\mathtt{let\ comp}\,\mathtt{x} = M\ \mathtt{in}\,F$ with jumps to remote locations $\mathtt{let\ dia}\,\mathtt{x} = M\ \mathtt{in}\,F$.

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

The logical content of these two rules is that possibility *subsumes* laxity. That is, the meaning we ascribe to the judgment A poss is weakened slightly to mean "A is true somewhere (original possibility) under some additional implicit conditions (laxity)." Similarly, the meaning of locally true hypotheses in Γ are weakened. However, this change is not adopted arbitrarily.

The logical scope of a computation $P \rightsquigarrow A$ is limited to a single world. The $\Diamond E$ rule remains unchanged, so we are required to discard all local assumptions in Γ (perhaps introduced with $\bigcirc E_p$) which were not promoted in some way to Δ. This reflects the notion that state, the implicit conditions underlying lax truth, cannot be carried over from one world to another.

We extend the language with addresses a^w which are the runtime values of type $\mathtt{ref}\,A$. Superscript w emphasizes the fact that addresses are a form of localized term. In the context of a store typing Θ associating addresses a^w to types A, we can describe the typing rules for primitive effects. Θ is omitted for clarity, except in typing rule *addr*.

$$\begin{array}{llcl} \text{Term} & M,N & ::= & \ldots \mid a^w \\ \text{Store Typing} & \Theta_w & ::= & \cdot \mid \Theta_w, a^w : A \end{array}$$

$$\frac{\Theta = \Theta_1, a^w : A, \Theta_2}{\Theta;\Delta;\Gamma \vdash a^w : \mathtt{ref}\,A}\ addr \qquad \frac{}{\Delta;\Gamma \vdash \mathtt{()} : \mathtt{1}}\ unit$$

$$\frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash \mathtt{ref}\,M \rightsquigarrow \mathtt{ref}\,A}\ talloc \qquad \frac{\Delta;\Gamma \vdash M : \mathtt{ref}\,A}{\Delta;\Gamma \vdash\ !M \rightsquigarrow A}\ tget$$

$$\frac{\Delta;\Gamma \vdash M : \mathtt{ref}\,A \quad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M := N \rightsquigarrow \mathtt{1}}\ tset$$

By limiting Θ to only those addresses bound in the local heap, we can ensure that dangling references are not typeable. The notion of a local heap is defined in section 5.3. Store addresses a^w are prohibited in mobile fragments of the program by the typing rules $\Box I$ and $\Diamond E$. This is achieved by treating store typing Θ similar to the local context Γ. Store typing does not interact directly with the other typing rules.

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

$$\begin{array}{llcll} \vdash & \lambda\mathtt{x} : \Box A\,.\,\mathtt{let\ box}\,\mathtt{u} = \mathtt{x}\ \mathtt{in}\,\mathtt{u} & \qquad & \vdash & \lambda\mathtt{x} : A\,.\,\mathtt{comp}\,[x] \\ : & \Box A \to A & & : & A \to \bigcirc A \\ \vdash & \lambda\mathtt{x} : \bigcirc A\,.\,\mathtt{dia}\,\{\mathtt{let\ comp}\,\mathtt{y} = \mathtt{x}\ \mathtt{in}\,[\mathtt{y}]\} & & \vdash & \lambda\mathtt{x} : A\,.\,\mathtt{dia}\,\{\mathtt{x}\} \\ : & \bigcirc A \to \Diamond A & & : & A \to \Diamond A \end{array}$$

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 $\Diamond A$.

5.2 Substitution

We can extend the prior notion of substitution to accommodate the new syntactic forms introduced by computations. Substitution of computations $\langle P/\mathtt{x}\rangle Q$ is also defined below, though not for primitive effects, whose semantics are not substitution-based. We omit the definition of $\langle P/\mathtt{x}\rangle F$ which is analogous to $\langle P/\mathtt{x}\rangle Q$.

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

5.3 Operational Semantics of Effects

To explain the semantics of mutable references, a local store H is added to each process l. Stores H_l are finite functions mapping addresses a^l to term values $\overline{V}$. The freely mobile terms $\langle r : M\rangle$ do not require a local store.

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

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.

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

For the fragment of the calculus relating to effects, we have general rules for sequential evaluation of computations, as well as some effect-specific primitives.

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

$$\frac{}{\langle l : H \vDash \mathtt{let\ comp}\,\mathtt{x} = \mathtt{comp}\ \mathtt{[}V\mathtt{]}\ \mathtt{in}\,F\rangle \Longrightarrow \langle l : H \vDash \langle\mathtt{[}V\mathtt{]}/\mathtt{x}\rangle F\rangle}\ \mathit{seq}_p$$

The definitions of contexts $\mathcal{S}$ and $\mathcal{S}^*$ allow us to reduce $\mathtt{let\ comp}\,\mathtt{x} = \mathtt{comp}\,P\,\mathtt{in}\,(\dots)$ to $\mathtt{let\ comp}\,\mathtt{x} = \mathtt{comp}\ \mathtt{[}\overline{V}\mathtt{]}\ \mathtt{in}\,(\dots)$. 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 or updating the store H with a binding

$[a^l \mapsto \overline{V}]$.

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

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

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

All reduction rules for effects are local and involve no communication. However, stores H are identified modulo equivalence of locations ($l \doteq l'$), so updates to H will affect all processes at the same location implicitly.

6 Extension with Recursion

We may also add recursion of various forms to the calculus, noting that such constructs are logically unsound and represent a departure from the Curry-Howard isomorphism. First and most obvious, we may introduce a term-level fixpoint construct. Secondly, we relax the restriction that dependencies among processes be acyclic, by giving a kind of global scope to some labels l denoting resources present in the environment. Such globally accessible labels permit recursion amongst processes, but this mechanism for designating locations global is not generally available to the programmer.

6.1 Fixpoint Constructs

We consider two natural forms of fixpoint corresponding to the distinction between variables $(\texttt{u} :: A)$ and $(\texttt{x} : A)$. We refer to $\texttt{fixv}\,(\texttt{u} :: A)\,.\,M$ as a valid or mobile fixpoint, and $\texttt{fix}\,(\texttt{x} : A)\,.\,M$ as local fixpoint. The operational semantics is given in the conventional way, with substitution used to perform unrolling. See section 11.2 of appendix.

$$\frac{\Delta, \texttt{u} :: A; \cdot \vdash M : A}{\Delta; \Gamma \vdash \texttt{fixv}\,(\texttt{u} :: A)\,.\,M : A}\; fix_v \qquad \frac{\Delta; \Gamma, \texttt{x} : A \vdash M : A}{\Delta; \Gamma \vdash \texttt{fix}\,(\texttt{x} : A)\,.\,M : A}\; fix$$

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 `fixv` or `fix`.

For example, $\texttt{fixv}\,(\texttt{u} :: \Diamond A)\,.\,\texttt{dia}\,E$ binds a fixpoint variable $(\texttt{u} :: \Diamond A)$ in E. The body expression E must be Γ-closed, a consequence of introducing $(\texttt{u} :: \Diamond A)$ rather than a local fixpoint variable $(\texttt{x} : \Diamond A)$. In E, the idiom $\texttt{let dia}\,\texttt{x} = \texttt{u}\;\texttt{in}\,F$ 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. It seems clear that mobile fixpoint over $(\texttt{u} :: \Diamond A)$ is a useful idiom. But local fixpoints $\texttt{fix}\,(\texttt{x} : \Diamond A)\,.\,\texttt{dia}\,E$ are not, since the scope of $(\texttt{x} : \Diamond A)$ is so limited. On the other hand, $\texttt{fix}\,(\texttt{x} : \bigcirc A)\,.\,\texttt{comp}\,P$ does seem useful for expressing recursion over a purely local computation P.

6.2 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 $\texttt{fixv}\,(\texttt{u} :: \Diamond A)\,.\,\texttt{dia}\,E$

require a set of assumptions ($\mathtt{v_i} :: \Diamond A_i$) in Δ, representing the locations amongst which a program can jump. It is not possible to realize such assumptions at the level of source programs, since labels l are runtime values with no syntax for their introduction. So an extra-logical means of obtaining or binding to these mobile values of type $\Diamond A$ is required. Furthermore, the acyclicity condition imposed on accessibility ψ must be weakened to permit mobile, *globally accessible* labels l.

In the context of this section, it is not possible to give a full account of globally accessible labels l and why such a concept is necessary. However, we can say that they arise out of a conflict between (1) our notion of accessibility constraints, which serve as a mechanism for imposing acyclicity on process configurations and (2) the desire for greater computational expressivity. Essentially, we must accommodate certain exceptional labels l that remain "accessible" (in a logical sense) despite movement from one process to another. For details, see the appendix, section 11.4.

7 Mobility and Concrete Datatypes

Just as our extension of the calculus with effects was motivated out of a desire to explore and explicate *immobility*, this extension with concrete datatypes is intended to reveal something about the division between mobile (or potentially mobile) values and immobile ones. It will be the case that values of some concrete types C have an inherent potential mobility, which is realized by a *marshalling* function $C \to \Box C$. Intuitively, such a function should convert a value into an equivalent mobile representation, though the specification $C \to \Box C$ alone does not ensure this.

We can introduce the usual logical type constructors for products and sums in the context of term and expression typing. All such connectives are essentially orthogonal to the notions of mobility (necessity) and locality (possibility). One can also add recursive types $\mu\alpha\,.\,B$ assuming a positivity restriction on α in B.

$$
\begin{array}{llcl}
\text{Type} & A, B & ::= & \ldots \mid \mathtt{1} \mid A * B \\
 & & & \mid \mathtt{0} \mid A + B \\
 & & & \mid \alpha \mid \mu\alpha\,.\,B \\
\text{Term} & M, N & ::= & \ldots \mid \mathtt{()} \mid \mathtt{(}M\mathtt{,}N\mathtt{)} \mid \mathtt{fst}\,M \mid \mathtt{snd}\,M \\
 & & & \mid \mathtt{inl}_A\,M \mid \mathtt{inr}_A\,M \\
 & & & \mid \mathtt{case}\,M\,\mathtt{of}\,\mathtt{inl}\,\mathtt{x} \Rightarrow N_1 \mid \mathtt{inr}\,\mathtt{y} \Rightarrow N_2 \\
 & & & \mid \mathtt{case}\,M\,\mathtt{of}\,\mathtt{inl}\,\mathtt{x} \Rightarrow F_1 \mid \mathtt{inr}\,\mathtt{y} \Rightarrow F_2 \\
 & & & \mid \mathtt{roll}_A(M) \mid \mathtt{unroll}(M)
\end{array}
$$

The typing rules for product ($A * B$) and sum types ($A + B$) are the usual ones. Introduction and elimination operate on local terms. For example, we are allowed to form a local tuple $\mathtt{(}M\mathtt{,}N\mathtt{)} : (A * B)$ when both components $M : A$ and $N : B$ are local terms. For simplicity, we use the usual iso-recursive formulation of μ-types with explicit $\mathtt{roll}_A()$ and $\mathtt{unroll}()$ operations.

$$\frac{\Delta;\Gamma \vdash M : A \quad \Delta;\Gamma \vdash N : B}{\Delta;\Gamma \vdash (M\,\texttt{,}N) : A * B}\ \textit{pair} \qquad \frac{\Delta;\Gamma \vdash M : A * B}{\Delta;\Gamma \vdash \texttt{fst}\, M : A}\ \textit{fst}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{()} : \texttt{1}}\ \textit{unit} \qquad \frac{\Delta;\Gamma \vdash M : A * B}{\Delta;\Gamma \vdash \texttt{snd}\, M : B}\ \textit{snd}$$

$$\frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash \texttt{inl}_{A+B}\, M : A + B}\ \textit{inr} \qquad \frac{\Delta;\Gamma \vdash M : B}{\Delta;\Gamma \vdash \texttt{inr}_{A+B}\, M : A + B}\ \textit{inl}$$

$$\frac{\Delta;\Gamma \vdash M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash N_1 : C \quad \Delta;\Gamma,\texttt{y} : A \vdash N_2 : C}{\Delta;\Gamma \vdash \texttt{case}\, M\, \texttt{of}\, \texttt{inl}\, \texttt{x} \Rightarrow N_1 \mid \texttt{inr}\, \texttt{y} \Rightarrow N_2 : C}\ \textit{case}$$

$$\frac{\Delta;\Gamma \vdash M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash Q_1 \rightsquigarrow C \quad \Delta;\Gamma,\texttt{y} : A \vdash Q_2 \rightsquigarrow C}{\Delta;\Gamma \vdash \texttt{case}\, M\, \texttt{of}\, \texttt{inl}\, \texttt{x} \Rightarrow Q_1 \mid \texttt{inr}\, \texttt{y} \Rightarrow Q_2 \rightsquigarrow C}\ \textit{case}_l$$

$$\frac{\Delta;\Gamma \vdash M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash F_1 \div C \quad \Delta;\Gamma,\texttt{y} : A \vdash F_2 \div C}{\Delta;\Gamma \vdash \texttt{case}\, M\, \texttt{of}\, \texttt{inl}\, \texttt{x} \Rightarrow F_1 \mid \texttt{inr}\, \texttt{y} \Rightarrow F_2 \div C}\ \textit{case}_p$$

$$\frac{\Delta;\Gamma \vdash M : [\mu\alpha\,.\,B/\alpha]B}{\Delta;\Gamma \vdash \texttt{roll}_{\mu\alpha\,.\,B}(M) : \mu\alpha\,.\,B}\ \textit{roll} \qquad \frac{\Delta;\Gamma \vdash M : \mu\alpha\,.\,B}{\Delta;\Gamma \vdash \texttt{unroll}(M) : [\mu\alpha\,.\,B/\alpha]B}\ \textit{unroll}$$

Definitions for natural numbers, booleans, and user-defined datatypes can be given in terms of products, sums, and recursive types, obviating the need to introduce such types as primitive notions. In the following sections, we explore the consequences of these definitions in terms of mobility and potential mobility.

7.1 Mobility

Values ($\texttt{box}\, M : \Box A$) have mobility by virtue of their form. This is enforced by the typing rule below. Intuitively, a mobile term has the property that it depends only on other mobile terms ($\texttt{u} :: A \in \Delta$).

$$\frac{\cdot;\Delta;\cdot \vdash M : A}{\Theta;\Delta;\Gamma \vdash \texttt{box}\, M : \Box A}\ \Box I$$

There is no *a priori* restriction on mobility based on the form of type A, so the calculus admits mobile functions $\Box(A \to B)$, products $\Box(A * B)$, etc. The type $\Box(\texttt{ref}\, A)$ has no values, but this is due to the fact that $\Box I$ excludes local variables ($\texttt{x} : A \in \Gamma$) and addresses ($a^l : B \in \Theta$) from scope.

Logically, we know the definition of mobility is not trivial, in the sense that $A \not\equiv \Box A$. Though $\Box A \to A$ holds for all A, the converse, $A \to \Box A$ is not provable for arbitrary A. We can see this by attempting to construct a normal-form proof of $\alpha \to \Box\alpha$, parametric in α:

$$\frac{\dfrac{\dfrac{\times}{\cdot;\cdot \vdash M : \alpha}\ ?}{\cdot;\texttt{x} : \alpha \vdash \texttt{box}\, M : \Box\alpha}\ \Box I(\alpha)}{\cdot;\cdot \vdash \lambda\texttt{x} : \alpha\,.\,\texttt{box}\, M : \alpha \to \Box\alpha}\ \to I \qquad \frac{\dfrac{\dfrac{}{\cdot;\cdot \vdash \texttt{()} : \texttt{1}}\ \textit{unit}}{\cdot;\texttt{x} : \texttt{1} \vdash \texttt{box}\, \texttt{()} : \Box\texttt{1}}\ \Box I}{\cdot;\cdot \vdash \lambda\texttt{x} : \texttt{1}\,.\,\texttt{box}\, \texttt{()} : \texttt{1} \to \Box\texttt{1}}\ \to I$$

The attempted derivation fails because we find there is no rule by which we can show $\cdot;\cdot \vdash M : \alpha$ for all α. But it could be the case that $A \to \Box A$ is provable for particular types

A or perhaps under some additional local assumptions. This is the case for $\mathtt{1} \to \Box\mathtt{1}$, via the derivation above. For some types A, the distinction between $\Box A$ and A collapses, due to the existence of a function $A \to \Box A$. We will now explore the significance and operational interpretation of this phenomenon.

7.2 Marshalling

Statically, a mobile value $\mathtt{box}\, M : \Box A$ is easy to distinguish from local one $V : A$, but local values of some types may have a kind of *potential* for mobility. This potential is realized via a function $A \to \Box A$ converting $V : A$ to a mobile value $\mathtt{box}\, M : \Box A$. One can view this operation as an abstract kind of *marshalling* — abstract because the type $A \to \Box A$ does not tell us everything about V or M and how they should be related. If we know V and M are observationally equivalent, this corresponds to the ordinary notion of marshalling as conversion to a mobile representation.

By considering these marshalling functions $A \to \Box A$, we can justify marshalling in the framework of modal types without resort to extra-logical marshalling primitives. This is good, because it allows us to judge the soundness or plausibility of marshalling at type $A * B$, $A + B$, $A \to B$, *etc.* based purely on type-theoretic considerations, independent of specific encodings for marshalled values. In many systems, marshalling is not integrated in a sound way, and the encoding/decoding process is partial. Failures occur when a non-mobile value is encountered during traversal of a value that is being marshalled. By considering only sensible, well-typed marshalling functions, we can avoid this class of errors.

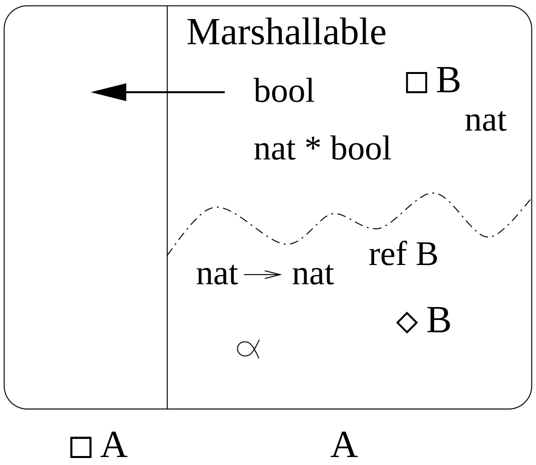

Figure 1: Mobility at a type. Some subset of values of type A may be convertible to $\Box A$ via marshalling, but others such as α (a value of arbitrary type) and $\mathtt{nat} \to \mathtt{nat}$ are not. The boundary may vary based on type-theoretic and functional requirements we impose on marshalling.

Potential mobility for values of a type is a somewhat fluid concept, being based on arguing the (non)existence of marshalling functions at that type. A function of type $A \to \Box A$ has the right type signature; given a value of type A, it produces some mobile value of type $\Box A$. But there are other conditions we should impose to ensure that the function implements marshalling correctly, or does not have certain undesirable properties. Hence there is uncertainty in drawing the dividing line between marshallable and non-marshallable values. Some natural conditions we might impose are as follows:

	A marshalling function f should...
Plausibility	satisfy $\Delta; \Gamma \vdash f : A \to \Box A$, ensuring that the resulting value has correct form.
Validity	satisfy $\Delta; \cdot \vdash f : A \to \Box A$, ensuring that the implementation is independent of local values $\mathtt{x} : B \in \Gamma$.
Total/Inductive	terminate when applied to finite, well-formed values.
Correctness	$V \equiv_o (\mathtt{let\ box\,u} = f\ V\ \mathtt{in\,u}) : A$ where $M \equiv_o N : A$ holds iff normal forms of M and N are observationally equivalent.

Regarding the correctness criterion, observational equivalence could be defined as a logical relation at all types including function types ($A \to B$). This generality is needed to establish correctness of a higher-order marshalling function such as:

$$\begin{array}{ll} \vdash & \lambda \mathtt{f} : A \to \mathtt{1} \,.\, \mathtt{box}\,(\lambda \mathtt{x} : A \,.\, ()) \\ : & (A \to \mathtt{1}) \to \Box(A \to \mathtt{1}) \end{array}$$

By considering marshalling in the abstract, we can understand how it interacts with various other features of a type system, and place upper and lower bounds on sensible implementations of marshalling. Obviously, weaker notions of marshalling will allow us to claim more types as marshallable and potentially mobile. We hope to understand what role each of the criteria play in the definition of marshalling. For the time being, we defer totality and correctness and argue for and against the existence of marshalling functions satisfying plausibility and validity criteria. We call the functions of type $A \to \Box A$ *plausible* marshalling functions.

7.2.1 Marshalling for Concrete Datatypes

Already, we have argued that a universal marshalling function $\alpha \to \Box\alpha$ (for all α) is not definable within the typed modal calculus. This is as it should be, since such a principle is logically unsound and trivializes the notion of location. Implemented as an external primitive, such a function could not be total in the presence of localized values. However, there is a specific class of types C for which we can exhibit plausible marshalling functions $C \to \Box C$.

Definition 1 (Concrete Datatypes) *Concrete types are formed from the following grammar. We require that that $\mu\alpha \,.\, C$ is used in a contractive way (excluding, for example, $\mu\alpha \,.\, \alpha$), and that α occur positively in C.*

$$\textit{Concrete Type}\quad C \quad ::= \quad 1 \quad | \quad C * C' \quad | \quad C + C' \quad | \quad \alpha \quad | \quad \mu\alpha \,.\, C$$

As an example, the natural numbers can be defined as a recursive datatype $\mu\alpha \,.\, 1 + \alpha$.

$$\begin{array}{rcl} \mathtt{nat} & = & \mu\alpha \,.\, 1 + \alpha \\ \mathit{zero} : \mathtt{nat} & = & \mathtt{roll}_{\mu\alpha \,.\, 1+\alpha}(\mathtt{inl}_{1+\alpha}\,()) \\ \mathit{succ} : \mathtt{nat} \to \mathtt{nat} & = & \lambda \mathtt{x} : \mathtt{nat} \,.\, \mathtt{roll}_{\mu\alpha \,.\, 1+\alpha}(\mathtt{inr}_{1+\alpha}\, x) \end{array}$$

A plausible marshalling function for type `nat` can be defined as follows. It is likely that this function meets the totality and correctness criteria as well, though this is not established.

```
fixv (m::nat -> □nat) . λ (x:nat) .
    case (unroll x) of
        inl(y) => box zero
        inr(y) => let box p = m(y) in box succ(p)
```

For all concrete types C, there is a plausible, valid marshalling function $C \to \Box C$. The following mutually recursive scheme defining $m_C : C \to \Box C$ gives the marshalling function for values of type C. We conjecture that this scheme satisfies totality and correctness criteria, for a suitable observational equivalence.

$$
\begin{array}{rcl}
m_{\mu\alpha\,.\,B} & = & \lambda \mathtt{x} : \mu\alpha\,.\,B\,.\,\mathtt{let\ box\,u} = m_{[\mu\alpha\,.\,B/\alpha]B}\ \mathtt{unroll(x)\ in\,(box\,roll}_{\mu\alpha\,.\,B}\mathtt{(u))} \\
m_{1} & = & \lambda \mathtt{x} : 1\,.\,\mathtt{box\ ()} \\
m_{A*B} & = & \lambda \mathtt{x} : A * B\,.\,\mathtt{let\ box\,u} = m_A\ \mathtt{(fst\,x)\,in\,let\ box\,v} = m_B\ \mathtt{(snd\,x)\,in\,box\ (u,v)} \\
m_{A+B} & = & \lambda \mathtt{x} : A + B\,.\,\mathtt{case\,x\,of\,inl\ y} \Rightarrow N_l \mid \mathtt{inr\,z} \Rightarrow N_r \\
 & & \text{where} \\
 & & N_l = \mathtt{let\ box\,u} = m_A\ \mathtt{y\,in\,(box\,inl}_{A+B}\,\mathtt{u)} \\
 & & N_r = \mathtt{let\ box\,u} = m_B\ \mathtt{z\,in\,(box\,inr}_{A+B}\,\mathtt{u)}
\end{array}
$$

The key property of concrete types which allows marshalling is that the introduction and elimination forms for product, sum, and recursive types are language extensions defined globally in a location-independent way. Consequently, a value $\mathtt{inl}_{A+B}\,V : A + B$ built at one location can be case-analyzed at another because everywhere $\mathtt{inl}_{A+B}$ is interpreted in the same way.

It is not always the case that values have such a location-independent meaning. Consider abstract datatypes, $\exists\alpha\,.\,B$, for example. Values of some unknown type α are created and manipulated via an interface B perhaps available only locally. It seems unlikely that a plausible, valid marshalling function could be defined, since the local interface B is the only means of creating and observing values of the abstract type α.

Handling marshalling of abstract types in a sound, abstraction-preserving manner is a well-known problem in distributed computation. The interaction between $\Box$, $\Diamond$, and quantification over types places some loose bounds on any solution we might adopt. The modal type system encourages us to consider a broader range of solutions and programming idioms, providing a context in which strategies from prior work can be understood and compared. This is discussed further in section 9.

Though marshalling functions can be implemented in the source language, the value of such an exercise lies in showing that values of some type A are logically independent of location, not in the behavior of such functions at runtime. The semantics is too high-level to reflect marshalling as conversion to a stream of bytes for transmission. But note that $\mathtt{box}\,M$ could be replaced easily with a lower-level representation, and only the $\mathtt{let\ box\,u} = M\ \mathtt{in}\ \ldots$ construct would be affected. Nothing else in the language allows us to access the underlying value of type A. We could, for example, choose a representation that makes the packaging of M for transmission more clear or exploits structure-sharing to reduce space requirements.

8 Examples

Using $\Box A$ and $\mathtt{let\ box\,u} = M\ \mathtt{in}\,N$, we can 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
```

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 [12].

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
        (* at server *)
        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
        (* at home *)
        let comp _ = h_env.write sv in
            {()}
     in

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

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`.

9 Proposed Plan of Future Work

It seems that we have a satisfactory account of how necessity and possibility from the logical realm are reflected operationally in the calculus, and how the calculus may be extended with effects, concrete data types, etc. Some extensions, such as mutable references, introduce localized terms which cannot be moved between locations — at least not in a way that (1) is consistent with the treatment of mobile terms of other types and (2) preserves well-formedness and semantic properties such as a reference's unique identity. Other extensions such as products $(A * B)$, sums $(A + B)$ and recursive types $(\mu\alpha \,.\, B)$ introduce location-neutral values, whose potential mobility depends only on the properties of their components. Abstract marshalling can be viewed as an interaction between these concrete forms of types C and mobility $\Box C$. We touched briefly on marshalling functions $C \to \Box C$, possible definitions and (non)existence in a few cases, but this development is by no means complete. There are also other forms of types whose interaction with mobility and locality we have yet to consider. Universal $\forall\alpha \,.\, B$ and existential $\exists\alpha \,.\, B$ quantification seem interesting, and may lead to new insights and techniques for assuring abstraction-safety in distributed computations.

9.1 Marshalling

There are some remaining questions on the theoretical side relating to the interplay of the static $(\mathtt{box}\, M)$ and dynamic (marshalling) notions of mobility. In section 7.2 we discussed marshalling functions of type $A \to \Box A$, and various possible criteria for declaring a function f to be a marshalling function.

We propose to develop a reasonable definition of marshalling incorporating some or all of plausibility, validity, totality, and observational equivalence criteria. Under this definition we will prove that certain families of functions implement marshalling correctly for the subset of concrete datatypes.

Though logical necessity $(\Box A)$ characterizes mobility, it is more difficult make claims about immobility, since it can only be defined negatively. As we have seen, local values of many types can be established to be mobile via marshalling $(A \to \Box A)$. If we relax the definition of marshalling, we find that encoding/decoding schemes $(\mathtt{encode} : A \to \Box B$ and $\mathtt{decode} : \Box B \to A)$ may exist for some types A and B, perhaps even in the absence of a simple marshalling function $A \to \Box A$. It is difficult to anticipate and analyze the full variety of protocols that might be used to transmit values from one location to another. We plan to focus on the simple notion of marshalling initially, perhaps considering other protocols if time allows.

9.2 Polymorphism

There are many logical axiom schemas and distributivity properties relating $\Box$, $\Diamond$ and other type constructors, which can be realized as proof terms using parametric polymorphism. Some of these, such as $\Box(\alpha \to \beta) \to (\Diamond\alpha \to \Diamond\beta)$ (remote application) and $(\Box\alpha * \Box\beta) \equiv \Box(\alpha * \beta)$ (marshalling a tuple) have an operational relevance. We plan to extend the calculus with polymorphism $\forall\alpha \,.\, B$ based on explicit type abstraction and application to permit programming with these sorts of generic combinators. Let-polymorphism with type inference would be another choice, but the inference problem is secondary to understanding the expressive power and interactions among quantification, $\Box$ and $\Diamond$.

When introducing type variables α and binding forms $\Lambda\alpha\,.\,M$ we face the question of whether or not to propagate the local (A `true`) mobile (A `valid`) distinction to the level of types. Framed logically, we need to decide how propositions (types) themselves interact with the worlds of a Kripke model. In many ways, it seems natural to assume propositions themselves have no location-dependence — that is, all types are sensible at all locations. This kind of a theory admits polymorphic marshalling functions, since formation of the type $\Box\alpha$ is only sensible if α is sensible at every accessible location.

It is important to note that a "global types" assumption does not imply that we abandon the distinction between local and mobile terms! For example, all locations "understand" the *type* `ref nat`, but individual *values* $a^w : \texttt{ref}\,\texttt{nat}$ are not portable between locations.

9.3 Abstract Types

As the dual of universal quantification, existential types $(\exists\alpha\,.\,B)$ raise the same issues regarding the appropriate logical scope of type variables and whether to distinguish between types of local and global scope. The spatial scope of type variables is important; this aspect of the theory determines whether two locations can ever share the *same* abstract type α, permitting values of type α to be exchanged. We present a few preliminary examples below indicating that the approach is promising, but formalization of the interaction of existentials with $\Box A$ and $\Diamond A$ is part of the proposed contribution of this thesis.

One possible treatment of existential types $\exists\alpha\,.\,B$ assigns type variables (names) local scope. Types quantified in this way are private to a single location, there being no analogue of marshalling at the level of types. We simply introduce assumptions $\alpha : \texttt{type}$ into the local context Γ. Types now have a well-formedness condition: $\overline{\Gamma} \vdash A : \texttt{type}$, where $\overline{\Gamma}$ consists of $\alpha_1 : \texttt{type}, \ldots, \alpha_k : \texttt{type}$.

$$\frac{}{\overline{\Gamma_1}, \alpha : \texttt{type}, \overline{\Gamma_2} \vdash \alpha : \texttt{type}} \quad \frac{\cdot \vdash A : \texttt{type}}{\overline{\Gamma} \vdash \Box A : \texttt{type}} \quad \frac{\cdot \vdash A : \texttt{type}}{\overline{\Gamma} \vdash \Diamond A : \texttt{type}}$$

$$\frac{\Delta; \Gamma \vdash M : [A/\alpha]B}{\Delta; \Gamma \vdash \texttt{pack}\,A\,\texttt{as}\,\alpha\,\texttt{in}\,M : \exists\alpha\,.\,B}\ \exists I$$

$$\frac{\Delta; \Gamma \vdash M : \exists\alpha\,.\,B \quad \Delta; \Gamma, \alpha : \texttt{type}, \texttt{x} : B \vdash N : C \quad \overline{\Gamma} \vdash C : \texttt{type}}{\Delta; \Gamma \vdash \texttt{unpack}\,M\,\texttt{as}\,(\alpha, \texttt{x})\,\texttt{in}\,N : C}\ \exists E$$

By adopting this theory of existential types, we essentially impose a per-location generativity on type names. Types such as $\Box\alpha$ and $\Diamond\alpha$ are not even well-formed, so we cannot begin to consider mobile values of type $\Box\alpha$ or marshalling functions $\alpha \to \Box\alpha$. Though a mobile abstract datatype `BoxAbsType` $: \Box\exists\alpha\,.\,B$ can be formed, every unpacked instance `TypeImpl` $: B[\alpha]$ is separate and not comparable to any other instance at the same or a different location.

```
let box AbsType = BoxAbsType in
unpack AbsType as (α,TypeImpl) in

let box u =
  box
    unpack AbsType as (β,TypeImpl) in
      (* α ≠ β *)
      ...
    end (* unpack *)
end (* unpack *)
```

Alternatively, one can choose to give existential type variables global scope, placing assumptions $\alpha :: \mathtt{type}$ in Δ. Valid types $\alpha :: \mathtt{type} \in \overline{\Delta}$ remain available underneath type constructors ($\Box$) and ($\Diamond$) when making the judgment $\overline{\Delta} \vdash A : \mathtt{type}$.

$$\frac{}{\overline{\Delta_1}, \alpha :: \mathtt{type}, \overline{\Delta_2} \vdash \alpha : \mathtt{type}} \quad \frac{\overline{\Delta} \vdash A : \mathtt{type}}{\overline{\Delta} \vdash \Box A : \mathtt{type}} \quad \frac{\overline{\Delta} \vdash A : \mathtt{type}}{\overline{\Delta} \vdash \Diamond A : \mathtt{type}}$$

$$\frac{\Delta; \Gamma \vdash M : [A/\alpha]B}{\Delta; \Gamma \vdash \mathtt{pack}\, A\, \mathtt{as}\, \alpha\, \mathtt{in}\, M : \exists \alpha \,.\, B} \; \exists I$$

$$\frac{\Delta; \Gamma \vdash M : \exists \alpha \,.\, B \quad \Delta, \alpha :: \mathtt{type}; \Gamma, \mathtt{x} : B \vdash N : C \quad \overline{\Delta} \vdash C : \mathtt{type}}{\Delta; \Gamma \vdash \mathtt{unpack}\, M\, \mathtt{as}\, (\alpha, \mathtt{x})\, \mathtt{in}\, N : C} \; \exists E$$

This corresponds to a second-order modal logic in which the set of well-formed propositions at a world increases monotonically with respect to accessibility. As noted before, this does *not* imply that values of type α are all mobile, merely that type α is sensible at all accessible locations. Such a theory permits forming types $\Box\alpha$ and, under some conditions, the exchange of abstract values between locations. Intuitively, this means that our locations share a common domain of types, so no matter what concrete type C implements α in $\mathtt{pack}\, \alpha\, \mathtt{as}\, C\, \mathtt{in}\, M : \exists \alpha \,.\, B$, that *type* C can be understood at any location.

Under this assumption, we can represent protocols for abstraction-safe marshalling as idioms within the modal calculus. Consider `AbsBoxType` $: \exists \alpha \,.\, \Box B$, denoting an abstract datatype with a mobile implementation $\Box B$. It is crucial that the implementation be mobile since a pure abstract value separated from its implementation is useless.

```
unpack AbsBoxType as (α,BoxImpl) in
let box I = BoxImpl in

(* α and I ``here''
   I.new : α
   I.insert : nat * α -> α
   I.marshall : α -> □ α
*)

let
  box a = I.marshall I.insert (42,I.new)
  box u =
    box
      (* α and I ``anywhere'' *)
      I.insert (0,a)
  ...
end (* unpack *)
```

Abstraction is not violated since the implementation ($\mathtt{I} :: B[\alpha]$) moves with the abstract value ($\mathtt{a} : \alpha$). The basic approach is sound, though perhaps not realistic. When the implementation depends on other bindings which themselves must be moved, the cost of moving an abstract value may grow quite large. This observation leads to the "hash types" solution of Leifer, Peskine, Sewell, and Wansbrough [13] in which module implementations are compared via a hash function applied to their source code. In this way, it is possible to avoid sending implementations for common modules stored in local libraries.

In cases where a simple marshalling function $\alpha \to \Box\alpha$ is not possible, one can implement transmission via encoding $\alpha \to \Box\beta$ and decoding functions $\Box\beta \to \alpha$, where α, β are existentially quantified. This resembles the strategy of the CLU language described by Herlihy and Liskov in [10]. The authors call datatypes obeying the protocol "transmissible clusters". Such datatypes have both an internal representation type (`rep`), and an external representation type (`xrep`). At runtime, the `decode` function examines the concrete type of a received value. Decoding succeeds if the value has the expected type (the implementation type of `xrep`). This assures type safety, but does not protect against violations of abstraction where two datatype implementations have the same `xrep` but different invariants. These errors are caught at runtime by the hash types scheme of Leifer, *et al.* They never arise in the modal calculus because implementations themselves are moved.

It should be noted that the modal calculus has a fully developed notion of mobility ($\Box A$) independent of the problem of abstraction-safe marshalling. The type theory is general enough to accommodate a variety of schemes for abstraction-safe marshalling. But other programming idioms can be explored also, including those involving abstraction and locality types $\Diamond A$. To boost expressivity even further, we could consider a mixed system with both local and global types (*kinds* `type` and $\Box$`type`) and corresponding introduction and elimination forms at the level of types. This would allow each abstract datatype to follow a separate policy governing generativity. We plan to explore the benefits of each approach through examples before committing to a particular theory of existential quantification.

We have discussed some possible ways to extend the calculus with existential types $\exists\alpha . B$ and operations `pack` and `unpack`. Once the tradeoffs between these approaches are understood, we plan to apply these insights to other treatments of abstract datatypes, specifically second-class modules with functors. More speculatively, we hope to demonstrate that effect typing $\bigcirc$ and mobility $\Box$ combine naturally to support abstraction-safe marshalling in the presence of effects (noting that hash types were not sound under effects [13]). If time allows, we propose to connect the modal type theory with the operational model of hash types to yield a more efficient, realistic semantics for code mobility.

9.4 Prototype Implementation

The modal calculus is interesting theoretically because it establishes a link between the logic of constructive $S4$ and distributed programming. But at a more pragmatic level, it guarantees a kind of marshalling-safety for distributed programs. Under the discipline of $\Box A$ and $\Diamond A$, we can be sure that the movement of terms amongst locations, is type safe, in the sense that a localized term is never interpreted out of its context (location). Furthermore, movement will not raise any "marshalling exceptions", since marshalling (at least at an abstract level) is accounted for statically within the type system. This is achieved without adopting a different interpretation of marshalling for localized values — creating remote references in these exceptional cases.

The benefits of the approach are that we have a uniform treatment of mobility at all types, including functions $\Box(A \to B)$, $\Box\Diamond A$, etc. The mobile values are those of type $\Box A$, (having the form $\mathtt{box}\, M$). This static "self-evident" mobility can be used to define (abstract) marshalling. Unlike the static notion of mobility, marshalling might *not* apply to all types, but we an opportunity to distinguish marshallable and non-marshallable types in a principled way. The cost of imposing a static type discipline is that the programmer is forced to be aware of mobility and locality, applying `box` and marshalling as necessary. As discussed above, polymorphism should reduce the burden of explicit marshalling to an acceptable level.

We propose to implement the mobility/locality type system for a small functional language, including effects, polymorphism, and abstract datatypes (in some form). Theoretical developments in the characterization of marshalling and treatment of type abstraction will guide implementation, while implementation will uncover various unrealistic assumptions in the high-level formulation of the semantics. For example, this interplay will help establish a relation between values of type $\Box A$ and their over-the-wire representation. At least initially, programs will run locally in an interpreter that simulates distributed execution and marshalling.

With simulation, we can explore how process scheduling, synchronization, and garbage collection would work. It seems to be the case that the restricted programming model derived from the $S4$ modal logic offers benefits in the area of garbage collection since cyclic dependencies can only occur when introduced through external, primitive means. Failure handling also seems to become simpler in some cases, since processes $\langle r : M \rangle$ are "stateless", and only processes $\langle l : H \vDash E \rangle$ have a persistent state. If time allows, we plan to explore these potential benefits.

Finally, the implementation will allow development of larger examples involving abstract datatypes and binding to remote resources. Through larger examples, we hope to show that the programming model of the calculus is useful in a practical setting, and that the effort required to program in a "location-aware" style is not too high relative to the benefit of type-safe mobility.

10 Summary and Plan

In summary, we have presented a calculus for distributed computation derived from constructive modal logic. The type system has its origins in a constructive formalization of S4 [18], and the operational semantics was inspired by an interpretation of logical worlds as sites for computation. Consistent with this logical heritage, the calculus satisfies type preservation and progress properties, and the core calculus (without fixpoints or effects) is known to enjoy strong normalization and confluence[3] [16].

We also showed that the core calculus is extensible and useful, in the sense that the modal type system captures mobility and locality of program terms. The extension to effectful computations (type $\bigcirc A$) demonstrated how the modal type discipline preserves locality of certain logically immobile term values — in our case, store addresses denoting mutable reference cells. The extension to concrete datatypes further illuminated the nature of the boundary between mobility and immobility. It seems clear that values of concrete type C are naturally location-neutral since the introduction and elimination forms for concrete, observable types allow one to implement marshalling ($C \to \Box C$) programmatically.

But the type-theoretic plausibility criterion ($C \to \Box C$) is not adequate to capture all the properties of marshalling as it is normally defined. We proposed various additional conditions in section 7.2. The plan is to develop these conditions formally so that certain marshalling functions can be proved to behave in the correct way, preserving not just the type C, but the observable properties of the marshalled value as well.

[3]The interested reader should see the appendix, section 11.1, and perhaps [16] for an explanation of the formal mechanisms used to represent locations and accessibility and to describe well-formed process configurations. These mechanisms play a crucial role in some of the proofs, but are not needed for a basic understanding of the calculus.

We also discussed plans for extending the language with polymorphism and abstract datatypes. We choose this line of inquiry in order to develop interesting applications and to better understand how mobility interacts with abstraction. Tentatively, it seems that one natural theory of abstract datatypes in the modal calculus allows encoding various protocols for abstraction-safe marshalling of values between locations. The operational model of exchanging underlying implementations may not be as realistic as those of other solutions, but the type system of the calculus does explain the phenomenon in a new light. Since the modal calculus has independent notions of mobility ($\Box$), locality ($\Diamond$), and effectful computation ($\bigcirc$), it seems likely that idioms exist for moving abstract values in more complex situations involving localized state or remote resources ($\Diamond$). Our plan is to find those idioms for distributed abstraction-safe programming or develop variations of the modal type theory to support them.

Finally, we plan to develop an implementation of a small functional language based on the type theory of mobility ($\Box A$) and locality ($\Diamond A$). The purpose of the implementation is to explore low-level refinements of the operational semantics, demonstrating how code mobility and distributed execution can be implemented in a feasible way. Though the prototype will not support compilation to low-level bytecode or machine code, most of the costs and problems involved in code mobility and marshalling will be exposed and resolved. The implementation will also serve as a platform for developing larger examples, so that limitations of the type system and programming model (beneficial or otherwise) become clear.

References

[1] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1989.

[2] Tijn Borghuis and Loe Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4), 2000.

[3] 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.

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

[5] 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.

[6] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. Technical Report MSR-TR-99-32, Microsoft, June 1999.

[7] 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.

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

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

[10] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst.*, 4(4):527–551, 1982.

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

[12] 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.

[13] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 87–98. ACM Press, 2003.

[14] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. *ACM Transactions of Programming Languages and Systems*, 25(1):1–69, January 2003.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (I & II). *Information and Computation*, 100(1):1–40 & 41–77, 1992.

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

[17] Álvaro Moreira. *A Type-Based Locality Analysis for a Functional Distributed Language.* PhD thesis, University of Edinburgh, 1999.

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

[19] 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.

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

11 Appendix

11.1 Well-Formed Process Configurations

To give a distributed operational interpretation to the calculus, we introduced labels w and a process notation $\langle w : \ldots \rangle$. An explanation of their static semantics was omitted in earlier sections in order to quickly establish an informal connection between the logical and operational readings of proof terms of the calculus. We will now extend our notion of typing in order to account for these new entities.

Location labels play a role similar to variables, but denote spatially remote terms or expressions residing in some other process. Labels r denote remote terms $\langle r : M \rangle$, and labels l denote remote expressions $\langle l : E \rangle$. Thus we treat r as a kind of hypothesis of validity, similar to $\mathrm{u} :: A$. Labels l will behave as a new kind of hypothesis of possibility. Runtime contexts Λ consist of a mixed collection of $r :: A$ and $l \div B$, representing the distributed context in which we judge a term or expression to be well-formed.

$$\text{Runtime Context} \quad \Lambda \quad ::= \quad \cdot \quad | \quad \Lambda, r :: A \quad | \quad \Lambda, l \div A$$

However, we must consider when, or more precisely from what locations we may make use of assumptions $r :: A$ or $l \div B$. In a distributed setting, the scope of labels is not lexically determined, rather it should be governed by a notion of *accessibility* between locations. This observation motivates the introduction of constraints ψ and judgment indices J.

$$\begin{array}{rcl} \text{Constraint} \quad \phi, \psi & ::= & \top \quad | \quad w \triangleleft w' \quad | \quad w \doteq w' \quad | \quad \phi \wedge \psi \\ \text{Location Index} \quad J & ::= & w \quad | \quad J\triangleleft \end{array}$$

The notation $\Lambda \backslash \psi$ is read as "Λ subject to ψ". Constraints ψ will determine which hypotheses in Λ are *accessible* from J, regarded as a location. Indices J specify 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.

The extended typing judgments are as follows: $\Lambda \backslash \psi; \Delta; \Gamma \vdash_J M : A$ is understood to mean that M is a term having 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 relevant judgment holds at all locations accessible from w. Thus logical validity is now associated with the form of judgment $\vdash_{w\triangleleft}$ in the presence of hypotheses Λ.

We first give the typing rules for labels w. The auxiliary judgment $\psi \vdash^a w \triangleleft w'$ means that w' is accessible from w under constraints ψ. Constraint entailment $\phi \vdash^a \psi$, presented in section 11.3 of the appendix, defines a small theory of accessibility ($w \triangleleft w'$) and equivalence ($w \doteq w'$) of locations.

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

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

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. An analogue of *ures* for labels l' would not be logically sound without symmetric accessibility.

Below 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. For the most part, indices J are propagated between premises and conclusion unchanged. But note that in $\Box I$ and $\Diamond E$, we use the quantified form of judgment $\vdash_{J\triangleleft}$ associated with logical validity (and thus mobility).

Core modal calculus:

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

$$\frac{}{\Delta, \mathtt{u} :: A, \Delta'; \Gamma \vdash_J \mathtt{u} : A}\ hyp^* \qquad \frac{\Delta; \Gamma \vdash_J M : A \to B \quad \Delta; \Gamma \vdash_J N : A}{\Delta; \Gamma \vdash_J M\ N : B} \to E$$

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

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

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

Generic effects:

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

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

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

Primitive effects:

$$\frac{\Theta = \Theta_1, a^w : A, \Theta_2}{\Theta; \Delta; \Gamma \vdash_w a^w : \mathtt{ref}\, A}\ addr \qquad \frac{\Delta; \Gamma \vdash_J M : A}{\Delta; \Gamma \vdash_J \mathtt{ref}\, M \div \mathtt{ref}\, A}\ talloc$$

$$\frac{\Delta; \Gamma \vdash_J M : \mathtt{ref}\, A}{\Delta; \Gamma \vdash_J\, !M \div A}\ tget \qquad \frac{\Delta; \Gamma \vdash_J M : \mathtt{ref}\, A \quad \Delta; \Gamma \vdash_J N : A}{\Delta; \Gamma \vdash_J M := N \div 1}\ tset$$

Datatypes:

$$\frac{\Delta;\Gamma \vdash_J M : A \quad \Delta;\Gamma \vdash_J N : B}{\Delta;\Gamma \vdash_J \texttt{(}M\texttt{,}N\texttt{)} : A * B}\ pair \qquad \frac{\Delta;\Gamma \vdash_J M : A * B}{\Delta;\Gamma \vdash_J \texttt{fst}\, M : A}\ fst$$

$$\frac{}{\Delta;\Gamma \vdash_J \texttt{()} : \texttt{1}}\ unit \qquad \frac{\Delta;\Gamma \vdash_J M : A * B}{\Delta;\Gamma \vdash_J \texttt{snd}\, M : B}\ snd$$

$$\frac{\Delta;\Gamma \vdash_J M : A}{\Delta;\Gamma \vdash_J \texttt{inl}_{A+B}\, M : A + B}\ inr \qquad \frac{\Delta;\Gamma \vdash_J M : B}{\Delta;\Gamma \vdash_J \texttt{inr}_{A+B}\, M : A + B}\ inl$$

$$\frac{\Delta;\Gamma \vdash_J M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash_J N_1 : C \quad \Delta;\Gamma,\texttt{y} : A \vdash_J N_2 : C}{\Delta;\Gamma \vdash_J \texttt{case}\, M\, \texttt{of inl x} \Rightarrow N_1 \mid \texttt{inr y} \Rightarrow N_2 : C}\ case$$

$$\frac{\Delta;\Gamma \vdash_J M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash_J Q_1 \rightsquigarrow C \quad \Delta;\Gamma,\texttt{y} : A \vdash_J Q_2 \rightsquigarrow C}{\Delta;\Gamma \vdash_J \texttt{case}\, M\, \texttt{of inl x} \Rightarrow Q_1 \mid \texttt{inr y} \Rightarrow Q_2 \rightsquigarrow C}\ case_l$$

$$\frac{\Delta;\Gamma \vdash_J M : A + B \quad \Delta;\Gamma,\texttt{x} : A \vdash_J F_1 \div C \quad \Delta;\Gamma,\texttt{y} : A \vdash_J F_2 \div C}{\Delta;\Gamma \vdash_J \texttt{case}\, M\, \texttt{of inl x} \Rightarrow F_1 \mid \texttt{inr y} \Rightarrow F_2 \div C}\ case_p$$

$$\frac{\Delta;\Gamma \vdash_J M : [\mu\alpha\,.\,B/\alpha]B}{\Delta;\Gamma \vdash_J \texttt{roll}_{\mu\alpha\,.\,B}(M) : \mu\alpha\,.\,B}\ roll \qquad \frac{\Delta;\Gamma \vdash_J M : \mu\alpha\,.\,B}{\Delta;\Gamma \vdash_J \texttt{unroll}(M) : [\mu\alpha\,.\,B/\alpha]B}\ unroll$$

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.

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

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.

11.2 Augmented Transition Rules

The transition rules of the semantics are now reformulated in such a way as to carry (and update) a set of constraints ψ. That is, a single step in the operational semantics becomes $C \setminus \psi \Longrightarrow C' \setminus \psi'$. 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 ($\psi \vdash^c C : \Lambda$).

$$
\begin{array}{llll}
\text{Term Context} & \mathcal{R} & ::= & [\,] \mid \mathcal{R}\ M \mid \overline{V}\ \mathcal{R} \mid \texttt{let box}\,\mathtt{u} = \mathcal{R}\,\texttt{in}\,N \\
 & & & \mid (\mathcal{R}, M) \mid (\overline{V}, \mathcal{R}) \mid \texttt{fst}\,\mathcal{R} \mid \texttt{snd}\,\mathcal{R} \\
 & & & \mid \texttt{inl}_{A+B}\,\mathcal{R} \mid \texttt{inr}_{A+B}\,\mathcal{R} \\
 & & & \mid \texttt{case}\,\mathcal{R}\,\texttt{of inl}\,\mathtt{x} \Rightarrow N_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow N_2 \\
 & & & \mid \texttt{roll}_A(\mathcal{R}) \mid \texttt{unroll}(\mathcal{R}) \\
\text{Computation Ctxt.} & \mathcal{S} & ::= & [\,] \mid [\mathcal{R}] \mid \texttt{let comp}\,\mathtt{x} = \mathcal{R}\,\texttt{in}\,Q \\
 & & & \mid \texttt{let box}\,\mathtt{u} = \mathcal{R}\,\texttt{in}\,Q \\
 & & & \mid \texttt{case}\,\mathcal{R}\,\texttt{of inl}\,\mathtt{x} \Rightarrow Q_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow Q_2 \\
 & & & \mid \texttt{ref}\,\mathcal{R} \mid\ !\mathcal{R} \mid \mathcal{R} := N \mid V := \mathcal{R} \\
 & & & \mid \texttt{let comp}\,\mathtt{x} = \texttt{comp}\,\mathcal{S}\,\texttt{in}\,Q \\
\text{Remote Comp. Ctxt.} & \mathcal{S}^* & ::= & [\,] \mid \{\mathcal{R}\} \mid \{\mathcal{S}\} \mid \texttt{let dia}\,\mathtt{x} = \mathcal{R}\,\texttt{in}\,F \\
 & & & \mid \texttt{case}\,\mathcal{R}\,\texttt{of inl}\,\mathtt{x} \Rightarrow F_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow F_2 \\
 & & & \mid \texttt{let box}\,\mathtt{u} = \mathcal{R}\,\texttt{in}\,F \mid \texttt{let comp}\,\mathtt{x} = \mathcal{R}\,\texttt{in}\,F \\
 & & & \mid \texttt{let comp}\,\mathtt{x} = \texttt{comp}\,\mathcal{S}\,\texttt{in}\,F
\end{array}
$$

The rules are presented in groups, according to structural commonalities.

Local $\quad \langle r : M \rangle \setminus \psi \Longrightarrow \langle r : M' \rangle \setminus \psi$

Rule	M		M'
app	$\mathcal{R}[\,(\lambda \mathtt{x} : A\,.\,M_1')\ \overline{V_2}\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V_2}/\mathtt{x}]M_1'\,]$
fst	$\mathcal{R}[\,\texttt{fst}\,(\overline{V_1}, \overline{V_2})\,]$	$\Longrightarrow$	$\mathcal{R}[\,\overline{V_1}\,]$
snd	$\mathcal{R}[\,\texttt{snd}\,(\overline{V_1}, \overline{V_2})\,]$	$\Longrightarrow$	$\mathcal{R}[\,\overline{V_2}\,]$
unroll	$\mathcal{R}[\,\texttt{unroll}(\texttt{roll}_A(\overline{V}))\,]$	$\Longrightarrow$	$\mathcal{R}[\,\overline{V}\,]$
caseL	$\mathcal{R}[\,\texttt{case}\,(\texttt{inl}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow N_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V}/\mathtt{x}]N_1\,]$
caseR	$\mathcal{R}[\,\texttt{case}\,(\texttt{inr}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow N_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V}/\mathtt{y}]N_2\,]$
fixpt$_v$	$\mathcal{R}[\,\texttt{fixv}\,(\mathtt{u} :: A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\![\texttt{fixv}\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]\!]M\,]$
fixpt	$\mathcal{R}[\,\texttt{fix}\,(\mathtt{x} : A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\texttt{fix}\,(\mathtt{x} : A)\,.\,M/\mathtt{x}]M\,]$

Local* $\quad \langle l : H \vDash E \rangle \setminus \psi \Longrightarrow \langle l : H \vDash E' \rangle \setminus \psi$

Rule	E		E'
app*	$\mathcal{S}^*[\,(\lambda \mathtt{x} : A\,.\,M_1')\ \overline{V_2}\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V_2}/\mathtt{x}]M_1'\,]$
fst*	$\mathcal{S}^*[\,\texttt{fst}\,(\overline{V_1}, \overline{V_2})\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,\overline{V_1}\,]$
snd*	$\mathcal{S}^*[\,\texttt{snd}\,(\overline{V_1}, \overline{V_2})\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,\overline{V_2}\,]$
unroll*	$\mathcal{S}^*[\,\texttt{unroll}(\texttt{roll}_A(\overline{V}))\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,\overline{V}\,]$
caseL*	$\mathcal{S}^*[\,\texttt{case}\,(\texttt{inl}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow N_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\mathtt{x}]N_1\,]$
caseR*	$\mathcal{S}^*[\,\texttt{case}\,(\texttt{inr}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow N_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\mathtt{y}]N_2\,]$
caseL*	$\mathcal{S}^*[\,\texttt{case}\,(\texttt{inl}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow Q_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow Q_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\mathtt{x}]Q_1\,]$
caseR*	$\mathcal{S}^*[\,\texttt{case}\,(\texttt{inr}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow Q_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow Q_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\mathtt{y}]Q_2\,]$
letdia*	$\texttt{let dia}\,\mathtt{x} = \texttt{dia}\,E\,\texttt{in}\,F$ (where $E \neq l'$)	$\Longrightarrow$	$\langle\!\langle E/\mathtt{x} \rangle\!\rangle F$
caseL$_p$	$\texttt{case}\,(\texttt{inl}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow F_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow F_2$	$\Longrightarrow$	$[\overline{V}/\mathtt{x}]F_1$
caseR$_p$	$\texttt{case}\,(\texttt{inr}_{A+B}\,\overline{V})\,\texttt{of inl}\,\mathtt{x} \Rightarrow F_1 \mid \texttt{inr}\,\mathtt{y} \Rightarrow F_2$	$\Longrightarrow$	$[\overline{V}/\mathtt{y}]F_2$
fixpt$_v^*$	$\mathcal{S}^*[\,\texttt{fixv}\,(\mathtt{u} :: A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\![\texttt{fixv}\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]\!]M\,]$
fixpt*	$\mathcal{S}^*[\,\texttt{fix}\,(\mathtt{x} : A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\texttt{fix}\,(\mathtt{x} : A)\,.\,M/\mathtt{x}]M\,]$

Interact		$C \setminus \psi \Longrightarrow C' \setminus \psi'$	
Rule	(C, C', ψ')		
letbox	$\langle r : \mathcal{R}[\,\texttt{let box}\,\texttt{u} = \texttt{box}\,M\,\texttt{in}\,N\,]\rangle$	$\Longrightarrow$	$\langle r' : M\rangle,\ \langle r : \mathcal{R}[\,[\![r'/\texttt{u}]\!]N\,]\rangle$
	where $\psi' = \psi \wedge (r' \lhd r) \wedge (\bigwedge_i \{r_i \lhd r' \mid \psi \vdash^a r_i \lhd r\})$		
*letbox**	$\langle l : H \vDash \mathcal{S}^*[\,\texttt{let box}\,\texttt{u} = \texttt{box}\,M\,\texttt{in}\,N\,]\rangle$	$\Longrightarrow$	$\langle r' : M\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,[\![r'/\texttt{u}]\!]N\,]\rangle$
letbox$_l$	$\langle l : H \vDash \mathcal{S}^*[\,\texttt{let box}\,\texttt{u} = \texttt{box}\,M\,\texttt{in}\,Q\,]\rangle$	$\Longrightarrow$	$\langle r' : M\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,[\![r'/\texttt{u}]\!]Q\,]\rangle$
letbox$_p$	$\langle l : H \vDash \texttt{let box}\,\texttt{u} = \texttt{box}\,M\,\texttt{in}\,F\rangle$	$\Longrightarrow$	$\langle r' : M\rangle,\ \langle l : H \vDash [\![r'/\texttt{u}]\!]F\rangle$
	where $\psi' = \psi \wedge (r' \lhd l) \wedge (\bigwedge_i \{r_i \lhd r' \mid \psi \vdash^a r_i \lhd l\})$		
syncr	$\langle r' : \overline{V}\rangle,\ \langle r : \mathcal{R}[\,r'\,]\rangle$	$\Longrightarrow$	$\langle r' : \overline{V}\rangle,\ \langle r : \mathcal{R}[\,\overline{V}\,]\rangle$
*syncr**	$\langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,r'\,]\rangle$	$\Longrightarrow$	$\langle r' : \overline{V}\rangle,\ \langle l : H \vDash \mathcal{S}^*[\,\overline{V}\,]\rangle$
	where $\psi' = \psi$		
syncl	$\langle l : H \vDash \texttt{let dia}\,\texttt{x} = \texttt{dia}\,l'\,\texttt{in}\,F\rangle,\ \langle l' : H' \vDash \overline{V^*}\rangle$		
	$\Longrightarrow \langle l : H \vDash l''\rangle,\ \langle l' : H' \vDash \overline{V^*}\rangle,\ \langle l'' : H' \vDash \langle\!\langle \overline{V^*}/\texttt{x}\rangle\!\rangle F\rangle$		
	where $\psi' = \psi \wedge (l' \doteq l'')$		

Note that only *syncl* and *letbox* (and variants) introduce new processes and hence must impose additional constraints ($\psi' = \psi \wedge \ldots$). In these cases we are essentially defining the location of the new process relative to existing processes.

Effects		$\langle l : H \vDash E\rangle \setminus \psi \Longrightarrow \langle l : H' \vDash E'\rangle \setminus \psi$			
Rule	H	E		H'	E'
seq	H	$\mathcal{S}^*[\,\texttt{let comp}\,\texttt{x} = \texttt{comp}\,[\overline{V}]\,\texttt{in}\,Q\,]$	$\Longrightarrow$	H	$\mathcal{S}^*[\,\langle[\overline{V}]/\texttt{x}\rangle Q\,]$
seq$_p$	H	$\texttt{let comp}\,\texttt{x} = \texttt{comp}\,[\overline{V}]\,\texttt{in}\,F$	$\Longrightarrow$	H	$\langle[\overline{V}]/\texttt{x}\rangle F$
alloc	H	$\mathcal{S}^*[\,\texttt{ref}\,\overline{V}\,]$	$\Longrightarrow$	$H \oplus [a^l \mapsto \overline{V}]$	$\mathcal{S}^*[\,[a^l]\,]$
get	H	$\mathcal{S}^*[\,!a^l\,]$	$\Longrightarrow$	H	$\mathcal{S}^*[\,[H(a^l)]\,]$
set	H	$\mathcal{S}^*[\,a^l := \overline{V}\,]$	$\Longrightarrow$	$H[a^l \mapsto \overline{V}]$	$\mathcal{S}^*[\,[()]\,]$

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 $[H]_\psi = [H']_\psi$.

11.3 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.

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

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 ($w \triangleleft w$) 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** ($\vdash_J \mathtt{u} : A$) and *poss* ($\vdash_J \{M\} \div A$). In fact, including a reflexivity axiom $\psi \vdash^a w \triangleleft w$ 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 $\phi_1, \phi_2, \ldots, \phi_n$.

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

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

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

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

11.4 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, $\forall \mathtt{w}\,.\,\mathtt{w} \triangleleft l$ 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.

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

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

The two inference rules are schematic in w allowing us to instantiate the quantifier with any world w. By design, there is no introduction form for $\forall \mathtt{w}\,.\,\mathtt{w} \triangleleft l$. The constraint $\forall \mathtt{w}\,.\,\mathtt{w} \triangleleft l$ is a primitive assertion about l which must be introduced explicitly.

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

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

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 $\Lambda \backslash \psi; \Delta; \cdot \vdash_{w\triangleleft} \mathtt{dia}\, l : \Diamond A$, which allows us to realize assumptions $\mathtt{u} :: \Diamond A$ in Δ.

11.5 Properties

Constraints ψ were introduced in section 11.1 and and entailment in 11.3 to describe the allowed dependencies between locations. We say that constraints ψ are *sound* if there are no cycles in accessibility constraints ($\psi \not\vdash^a w \triangleleft w$).

Definition 2 *A constraint formula ψ is sound (ψ sound) iff $\nexists w\ .\ \psi \vdash^a w \triangleleft w$.*

Our type preservation theorem states that the transition rules preserve types for processes and soundness of constraints.

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. See also [16].

Case:

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

$\Lambda\backslash\psi \vdash^s_l H : \Theta$	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
$\cdot; \Lambda\backslash\psi; \cdot; \cdot \vdash_{r'\triangleleft} \overline{V} : B$	Assumption, Definition
$\psi \vdash^a r' \triangleleft l$	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:

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

$\Lambda\backslash\psi \vdash^s_l H : \Theta$	Assumption, Definition
$\Theta; \Lambda\backslash\psi; \cdot; \cdot \vdash_l \mathcal{S}^*[\,\mathtt{let\ box}\,\mathtt{u} = V\ \mathtt{in}\, N\,] \div C$	Assumption, Definition
$\Theta; \Lambda\backslash\psi; \cdot; \cdot \vdash_l \mathtt{let\ box}\,\mathtt{u} = V\ \mathtt{in}\, N : B$	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 ($\Box E$)
$\cdot; \Lambda\backslash\psi; \cdot; \cdot \vdash_{l\triangleleft} M : A$	Inversion ($\Box I$)
Let $\Lambda' = \Lambda, r' :: A$	
$\psi' = \psi \wedge (r' \triangleleft l) \wedge (\bigwedge_i \{r_i \triangleleft r' \mid \psi \vdash_w r_i \triangleleft l\})$	Assumption
$\psi \vdash^a \phi \implies \psi' \vdash^a \phi$	Entailment $\vdash^a$
$\psi' \vdash^a r_i \triangleleft l \implies \psi' \vdash^a r_i \triangleleft r'$	Entailment $\vdash^a$
$\psi' \vdash^a r' \triangleleft l$	Entailment $\vdash^a$
$\cdot; \Lambda'\backslash\psi'; \cdot; \cdot \vdash_{r'\triangleleft} M : A$	Mobility Against Accessibility
$\Theta; \Lambda'\backslash\psi'; \cdot; \cdot \vdash_{l\triangleleft} r' : A$	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
$\exists w, w'\ .\ \psi' \vdash^a w \triangleleft w'$ contradicts ψ `sound`	Entailment $\vdash^a$

ψ' `sound`	By Contradiction
$\Lambda' \supseteq \Lambda$	Directly

Case:

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

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

□

The progress theorem states that a well-formed process configuration is either terminal or is reducible with some transition rule. Note that soundness (acyclicity) is critical because it rules out deadlocks such as $\langle r : r'\ V\rangle, \langle r' : r\ V'\rangle$. Thus we must exclude globally accessible labels, though term-level fixpoints are compatible with progress.

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 [16]. The same strategy extends to the fragment with effects.

The operational semantics we have presented is non-deterministic in that there is often a choice of which process to reduce and, within a process, a choice of when to perform optional synchronization steps. However, in [16], we establish that the core calculus (without effects or recursion) satisfies strong normalization and confluence (modulo deferred synchronization steps).