Thesis Proposal:
Type Theory for Mobility and Locality

Jonathan Moody
jwmoody@cs.cmu.edu

January 19, 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.* [11] and $lsd\pi$ by Ravara, Matos, *et al.* [20]. 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 [12]. 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.

Local proxies for remote objects provide "stub" methods to forward these method calls to the remote object. The proxy class implements marshalling of parameters and unmarshalling of a return value or exception. Subtyping on classes and interfaces essentially hides the distinction between local and remote objects. Both the real 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.

The mechanisms for obtaining a proxy for a remote object, and moving values among locations are not connected. Typically a program obtains references to remote objects by specifying some form of external identifier for that object; the identity and location of the remote instance depends on how the lookup operation is implemented. Movement is handled as follows. Certain primitive values and all objects derived from classes implementing the Serializable interface are marshallable by copying. 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 [21], 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 $\Delta; \Gamma \vdash A$ `true` and $\Delta; \Gamma \vdash A$ `poss`, where Δ are assumptions A `valid`, and Γ are assumptions A `true`.

The intuition behind our application of modal logic to distributed programming is the following: If we interpret propositions as types and proofs as programs, it is also quite natural to interpret the logical worlds as sites for computation. Furthermore, we see that validity corresponds to mobility or portability of terms between locations, and possibility corresponds to 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.

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

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.

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

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.

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

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

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.

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

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

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 $\Delta; \Gamma \vdash M : A$.

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

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`.

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

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.

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

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 12.1, 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.

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

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

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

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

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:*

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

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.

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

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 ($\texttt{let box u = box } M \texttt{ in } \ldots$), 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.

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

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 [13]. 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 $\texttt{let dia x = dia}\, E\, \texttt{in}\, F$, we simply substitute the computation E for $\texttt{x}$ in F using expression substitution $\langle\!\langle E/\texttt{x}\rangle\!\rangle F$. This operation rearranges the structure of the computation locally; no actual movement between locations occurs. The restriction that E not have the form of a label l' is crucial because substitution of a label $\langle\!\langle l'/\texttt{x}\rangle\!\rangle F$ does not allow us to make progress.

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

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 $\texttt{x} : A$ which will be provided at the destination.

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

Duplication of $\overline{V*}$ from process $\langle l' : \overline{V*}\rangle$ as $\langle l'' : \langle\!\langle \overline{V*}/\texttt{x}\rangle\!\rangle F\rangle$ is needed to assure type preservation in cases when more than one process might jump to l'. We 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 12.3.

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 12.1 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 $\texttt{x} : A$. 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 $\texttt{box}\,M$ and $\texttt{let box u} = M\,\texttt{in}\,N$. 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 $\texttt{box}\,M$ is mobile if M is Γ-closed. There is no particular location where M must be evaluated, since it has been established that $\texttt{box}\,M : \Box A$. 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, $\texttt{dia}\,E$, $\texttt{let dia x} = M\,\texttt{in}\,F$, 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 $\texttt{dia}\,E : \Diamond A$ refer to trivially remote resources. If we admit $\texttt{u}_\texttt{i} :: \Diamond A_i$ bound to values $\texttt{dia}\,l_i$, 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.

[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.

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, 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]

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

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

We introduce a new form of judgment $P \mathrel{\sim\!\!\!>} A$, 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 ($\mathtt{comp}\,P$) requires that P be a purely local computation ($P \mathrel{\sim\!\!\!>} A$). Operationally, the elimination form $\mathtt{let\ comp\,x = comp}\,P\,\mathtt{in}\,Q$ corresponds to sequential evaluation of P followed by Q, binding the result of P to local variable $(\mathtt{x} : A)$

[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 9, for example.

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.

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

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

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 $\mathtt{x} : A$, *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 $\mathtt{let\ dia}\, \mathtt{x} = M\, \mathtt{in}\, F$.

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

The logical content of these two rules is that possibility *subsumes* laxity. That is, the meaning we ascribe to the judgment $A\ \mathtt{poss}$ 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 $P \leadsto A$ 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*. Θ does not interact directly with the other typing rules.

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

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

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

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

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.

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

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

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 $\langle P/\mathtt{x}\rangle Q$.

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

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

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.

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

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.

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

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.

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

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

The definitions of contexts $\mathcal{S}$ and $\mathcal{S}^*$ allow us to reduce $\texttt{let comp}\,\texttt{x} = \texttt{comp}\,P\,\texttt{in}\,(\dots)$ to $\texttt{let comp}\,\texttt{x} = \texttt{comp}\,[\overline{V}]\,\texttt{in}\,(\dots)$. 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 $[a^l \mapsto \overline{V}]$.

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

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

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

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 $(\mathtt{u} :: A)$ and $(\mathtt{x} : A)$. We refer to $\texttt{fixv}\,(\mathtt{u} :: A)\,.\,M$ as a valid or mobile fixpoint, and $\texttt{fix}\,(\mathtt{x} : A)\,.\,M$ as local fixpoint. The operational semantics is given in the conventional way, with substitution used to perform unrolling. See section 12.2 of appendix.

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

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

For example, $\texttt{fixv}\,(\mathtt{u} :: \Diamond A)\,.\,\texttt{dia}\,E$ binds a fixpoint variable $(\mathtt{u} :: \Diamond A)$ in E. The body expression E must be Γ-closed, a consequence of introducing $(\mathtt{u} :: \Diamond A)$ rather than a local fixpoint variable $(\mathtt{x} : \Diamond A)$. In E, the idiom $\texttt{let dia}\,\mathtt{x} = \mathtt{u}\,\texttt{in}\,F$ 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 $(\mathtt{u} :: \Diamond A)$ is a useful idiom. But local fixpoints $\texttt{fix}\,(\mathtt{x} : \Diamond A)\,.\,\texttt{dia}\,E$ are not, since the scope of $(\mathtt{x} : \Diamond A)$ is so limited. On the other hand, $\texttt{fix}\,(\mathtt{x} : \bigcirc A)\,.\,\texttt{comp}\,P$ 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 $(\texttt{v}_\texttt{i} :: \Diamond A_i)$ in Δ, representing the locations amongst which a program can jump.

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 12.4.

7 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 reveals something about the division between mobile (or potentially mobile) values and immobile ones.

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 notion of necessity and possibility. One can also add recursive types $\mu\alpha\,.\,B$ assuming a positivity restriction on α in B.

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

The typing rules for product $(A * B)$ and sum types $(A + B)$ are the usual ones, with unit elements `1` and `0`, respectively. For simplicity, we use the usual iso-recursive formulation of μ-types with explicit $\texttt{roll}_A()$ and $\texttt{unroll}()$ operations.

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

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

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

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

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

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

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 section, we use concrete datatypes to penetrate to the center of the question of mobility, namely what distinguishes *immobile* values from those which are potentially mobile?

8 Remarks on (Im)mobility

The typing rules $\Box I$ and $\Diamond E$ play a crucial role in the static semantics of the calculus. The natural operational reading of these rules relates them to locality and potential mobility of terms. It is tempting to claim that $\Box A$ represents mobility (available every place), and $\Diamond A$ represents immobility (available some place). But the judgments $M : A$ and $M : \Diamond A$ do not represent the *complement* of mobility, rather situations in which M is local or remote, but not *known* to be mobile. Thus *immobility* is not a fundamental concept, but is derived as the complement of mobility. The judgment $M : A$ does not preclude mobility of M, since it could also be the case that $\mathtt{box}\, M : \Box A$ or there may exist a function $f : A \to \Box A$ defined on the value of M. The latter case motivates the following definition:

Definition 1 (Marshalling Functions) *A function term $F = \lambda \mathtt{x} : A\,.\,M$ is a strong marshalling function at type A iff $;\vdash F : A \to \Box A$ and for all V of type A, $F\ V \to^* \mathtt{box} V$. In cases when $FV \to^* \mathtt{box} V'$ for $V \equiv V'$ we say F is a marshalling function modulo the equivalence $\equiv$.*

So mobility at type A is not strictly determined by the syntactic form $\mathtt{box}\, M : \Box A$, but is a property of the type A and the totality of operations defined on A. We will see that concrete, observable data types are strongly marshallable, as are values of type $\Box A$, but $A \to B$ and $\Diamond A$ are not (generally) marshallable.[3]

Let us first consider the forms of value in the core calculus — those of type $\Box A$, $A \to B$, and $\Diamond A$. A marshalling function for type $\Box A$, can be given as:

$$m_{\Box A} \quad = \quad \lambda \mathtt{x} : \Box A\,.\,\mathtt{let}\ \mathtt{box}\, \mathtt{u} = \mathtt{x}\ \mathtt{in}\, (\mathtt{box}\, \mathtt{box}\, \mathtt{u}) : \Box A \to \Box\Box A$$

[3] An investigation of abstract types is ongoing and seems likely to be another source of immobility.

This is actually one of the characteristic axioms of S4. The implementation above defines a weak marshalling function modulo evaluation, since our semantics permits evaluation under `box`. That is, $\texttt{box}\, M$ will be mapped to $\texttt{box}\,\texttt{box}\, r$ where r denotes the value of M.

In general, it seems that no marshalling function exists for values of type $A \to B$ or $\Diamond A$ for arbitrary A, B. But immobility is a slippery concept and there are exceptions for certain finitely enumerable types. For example, a function $\Diamond\texttt{1} \to \Box\Diamond\texttt{1}$ exists:

$$m_{\Diamond A} \quad = \quad \lambda \texttt{x} : \Diamond\texttt{1} \,.\, \texttt{box}\,\texttt{dia}\,\{()\} : \Diamond\texttt{1} \to \Box\Diamond\texttt{1}$$

A marshalling function for $(\texttt{bool} \to \texttt{bool}) \to \Box(\texttt{bool} \to \texttt{bool})$ is also definable, since one can apply the function to the elements `true` and `false`, noting the results, and construct an equivalent mobile function from this truth table representation. But for arbitrary values $A \to B$ and $\Diamond A$, there may be no such trick we can use to observe its essential properties programmatically, and hence no way to implement marshalling.

We note that providing marshalling as an external non-logical primitive $(A \to B) \to \Box(A \to B)$ is problematic at best. The type $A \to B$ does not reflect the environment in which the closure value was formed, and this environment could contain bindings of any type. Essentially, we are then obligated to provide marshalling for *all* values in the language, or admit the possibility of runtime marshalling errors.

In the case of $\Diamond A$, our means of observing such values is also quite restrictive; only `let dia` permits examination of the underlying value. As with function closures, values of type $\Diamond A$ may be closed under an unknown environment, making static analysis of marshalling difficult. It is interesting to note that constructive S5 includes a logical axiom $\Diamond A \to \Box\Diamond A$, a kind of marshalling function for $\Diamond A$. This is not at all contradictory with our prior observation. Since S5 relies on an assumption of symmetric accessibility, we are not *required* to actually marshal the closure representation of $\texttt{dia}\, E : \Diamond A$ by copying. Rather we are free to use a proxy. The proxy can be moved to any accessible world, while the underlying value remains accessible by symmetry.

8.0.1 Marshalling for Concrete Datatypes

Definition 2 (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.*

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

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

$$\begin{aligned} \texttt{nat} \quad &= \quad \mu\alpha\,.\,\texttt{1} + \alpha \\ \textit{zero} : \texttt{nat} \quad &= \quad \texttt{roll}_{\mu\alpha\,.\,\texttt{1}+\alpha}(\texttt{inl}_{\texttt{1}+\alpha}\,()) \\ \textit{succ} : \texttt{nat} \to \texttt{nat} \quad &= \quad \lambda \texttt{x} : \texttt{nat}\,.\,\texttt{roll}_{\mu\alpha\,.\,\texttt{1}+\alpha}(\texttt{inr}_{\texttt{1}+\alpha}\, x) \end{aligned}$$

A strong marshalling function for type `nat` is given as:

```
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)
```

We conjecture that for all concrete types C, there is a canonical implementation of a strong 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. Termination depends on the finite size of values of type C.

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

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

9 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 $\mathtt{let\ box\,u} = \mathtt{box}\,M\ \mathtt{in}\ \ldots$ is an idiom for spawning M for parallel evaluation, similar to `(let (u (future` M`)) ...)` in Multilisp [13].

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

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

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

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

```
let dia h_env = home in
```

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

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

The use of ◊ and/or ◯ 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`.

10 Proposed Plan of Future Work

It seems that we have a satisfactory account of how various features from the logical realm are reflected operationally in the calculus, and how the calculus may be extended with effects, concrete data types, etc. There are some remaining questions on the theoretical side relating to polymorphism and abstract types and an examination of their interaction with mobility. Through this, we hope to shed light on the origin of immobility, showing that it can be explained in a clean type-theoretic way.

We also plan to embark on an implementation of the calculus in the ConCert framework for grid computing [9], culminating with a useable prototype. We believe that the programming model of the calculus is suitable for grid programming since the $\rightarrow, \Box$ fragment of the calculus is at the core of Hemlock, the current prototype language and compiler. We see an opportunity to expand the programming model of Hemlock, mixing spawned computations with access to resources at definite locations, through the use of the full spectrum of type constructs $\rightarrow, \Box, \Diamond$ (and extensions) available in the calculus of modal logic. We concede that parts of the language (particularly explicit marshalling) will have to be adapted to make it more palatable.

10.1 Theoretical Issues to be Addressed

10.1.1 Abstract Types

In prior sections, we defined canonical marshalling functions for all concrete datatypes. All such concrete values are location-neutral, and we may marshal them at any time. For abstract types, the situation is more complicated. We must distinguish between $\exists\alpha\,.\,B$ (a package or module) and the abstract implementation type hidden from clients by means of type variable α.

Consider the case of $\texttt{box}\, M : \Box\exists\alpha\,.\,B$. If our interpretation of the type constructors is to remain modular and logical, we are forced to conclude that this particular package $M : \exists\alpha\,.\,B$ is mobile, since it closed and constructed solely from other mobile terms. So it is not the case that $\exists\alpha\,.\,B$ is always immobile, rather, we conjecture that it is the type abstraction over α that is the source of immobility for values of type α.

There is a very good argument why one would *want* such values to be immobile. Abstraction is a means of protecting the representation of a value from outside observation and ensuring that clients use a certain interface to manipulate that value. This may be important for reasons of privacy or modularity. Though the hidden type implementing α may be known to the compiler or the runtime system, it would violate the spirit of the abstraction to *separate* a value of type α from the package $\exists\alpha\,.\,B$. This is why the usual package open rule is formulated in the way it is, to ensure that nothing involving type α escapes to the surrounding scope.

As with types $A \to B$ and $\Diamond A$, one cannot *disprove* the existence of marshalling functions for all such types. In some cases it will be possible to observe certain aspects of values of type α programmatically and encode these properties as concrete, marshallable values of type C. By composing encoding $\exists\alpha\,.\,B \to C$ with marshalling $C \to \Box C$, a programmer might recreate an "equivalent" value $\exists\alpha\,.\,B$ elsewhere. But at the same time, there are good arguments why no generally applicable, canonical marshalling function exists for $\exists\alpha\,.\,B$. In general, arguing equivalence for two abstract datatypes requires some external specification of the semantics of that type, and when encoding is left up to the programmer, we cannot make any guarantees about the behavioral properties of the encoding scheme.

We might also link the locality of effectful computation with abstraction. Essentially, the type $\bigcirc A$ is implicitly quantified over some representation of the machine state, and we can think of $\texttt{ref}\, A$ as carrying a subscript $\texttt{ref}_\alpha\, A$. There are certain primitive operations are defined on the abstract type of computations, but the internal machine state α remains inaccessible. However, it is possible to partially observe some aspects of the state. For example, with mutable references (without equality), it is still possible to observe the contents of a reference. This allows one to recreate a reference anywhere, though of course the identity and structure sharing of such a replica is not preserved. Given an equality test on reference cells, one could implement a stronger marshalling function which preserves structure sharing among some set of references.

10.2 Towards Implementation

In section 3, we presented an operational semantics for the calculus based on concurrent reduction of program fragments. Process notation ($\langle r : M\rangle$ and $\langle l : E\rangle$) was used to reflect the placement of terms M or expressions E at abstract locations r or l. The internal behavior of processes is not of much concern to us, since standard compilation strategies for functional languages will apply. Rather the problems lie in implementing interactions between processes. In our formulation of the semantics, these interactions occur in the *syncr*, *syncr** and *syncl* rules.

By design, processes $\langle r : M\rangle$ can be scheduled to run at an arbitrary location. At the same time, processes holding references to r must be able to synchronize on the value of r either by locating the process itself, or obtaining the result value in another way. In either case, some means of indirection is needed. On the other hand, our operational semantics treats processes $\langle l : E\rangle$ as resources at fixed locations. We note that the number of such distinct locations $l \not\equiv l'$ remains fixed over the lifetime of program execution, so the treatment of labels l may not require the full generality of the solution for labels r.

The primary factor determining runtime performance will be communications latency, setting aside the orthogonal issue of proof-checking transmitted code. Any reorganization of

the semantics should seek to reduce latency of operations, insure that the method is scalable to large graphs of processes, and increase robustness in the presence of failures.

We propose to deepen and, to an extent, reorganize the semantics to make implementation feasible. Some experimentation might be required in exploring the design space since there are many strategies for handling synchronization more efficiently and allowing garbage collection of processes. Essentially we would like to maintain as much as possible a local picture of those aspects of the global state with which a process interacts.

The main defect of the abstract semantics is the handling of synchronization on the result values of processes r. For simplicity, we used processes $\langle r : V \rangle$ to represent a sort of distributed environment mapping labels r to values V, not specifying any means to delete processes or memoize result values near the locations they might be accessed. Furthermore, the mechanism for assigning processes to a machine and looking up the assigned location of a process is omitted. Finally, the abstract semantics assumes that processes persist indefinitely in order to represent the "binding" of r to V as $\langle r : V \rangle$. Preserving processes r at a remote locations is not efficient or realistic given the high latency of communication and the possibility of node failure.

10.2.1 Resource Binding and Discovery

As hinted earlier, we may interpret values of type $\Diamond A$ as denoting resources present in the distributed environment. Specifically, a value $\texttt{dia}\, l_i : \Diamond A$ represents a resource of type A at a location l_i. One can envision a variety of mechanisms for binding to such resources, including explicit resolution of resources by their name ($\texttt{lookup}_A :: \texttt{resource\_name} \rightarrow \Diamond A$), or an implicit linking phase in which unbound variables $\texttt{u} :: \Diamond A$ are bound automatically to a set of locations.

For some applications, the identity of location l_i will be relevant and resolution by name should be used. For example, a programmer might want to access a particular dataset stored at a remote site. By providing a name or address, the programmer designates which dataset or location is intended. In other cases, the binding mechanism may choose arbitrarily among some set of locations in the distributed environment providing a resource of type A. This form of binding might be used to locate host machines willing to play a certain generic role or provide some service during a computation. One could envision a small set of generic service types such as $\Diamond\texttt{CONSOLE}$, providing interaction with the user who submitted the program for execution, or $\Diamond\texttt{FILE\_IO}$, providing an interface to scratch storage space.

Such generic resources could be advertised and the discovery process piggybacked onto the protocol by which worker nodes are discovered. The ConCert conductor software will have to be modified to accept input from the owner of the host machine as to which set of libraries or services to advertise. Interestingly, the implementation of a service or resource may or may not be provably safe under the safety policy, though perhaps it is trusted by the machine owner. If the owner chooses to run potentially unsafe code to provide a service, this has no impact on the broader "trustless" security model of ConCert.

10.2.2 Labels and Concrete Locations

The current formulation of the semantics presents some difficulties for efficient implementation. Firstly, we must consider the abstract locations w and their mapping to concrete network nodes. The rules concerning interaction between processes (*syncr*, *syncr**, and *syncl*) assume that we are able to *locate* an arbitrary process w at will. For example, *syncr* assumes that we can locate the process $\langle r' : \overline{V} \rangle$ in order to retrieve the value $\overline{V}$ (or wait for a value V to become available). There are a variety of solutions:

- Static: Replace labels w with concrete addresses. This avoids any bottlenecks associated with indirect lookup. It is not immediately compatible with a work-stealing model, in which process creation is decoupled from assignment to a concrete network node.
- Centralized lookup: A table mapping each w to a concrete location is maintained at some distinguished "home" process, probably at the location where the program was submitted for execution.
- Distributed lookup: Each process or location maintains a partial table, and these mappings are used to shortcut lookup in the remote table(s) managed by ancestors. In the average case, this should be better than centralized lookup. If process mobility or rebinding of abstract locations is allowed, then naturally an update or invalidation protocol is also required.

The two kinds of label r and l could be managed in the same way, or differently. For example, we may want to use indirect lookup for r, but direct addressing for l. This would be the case if processes r are assigned locations by work-stealing, but processes l are bound to fixed locations during a linking phase. But note that the notions of a "parent" for locations l and r differ. The parent of a process r is a process r' or l' such that $r \triangleleft r'$ or $r \triangleleft l'$. The parent of a process l is a process l' such that $l' \triangleleft l$. In a process r we need to resolve references to r', and in a process l we need to resolve both r' and l'. In both cases, a distributed lookup scheme will function correctly if we look to the parent to resolve w, assuming the table at the "home" location is populated with mappings for known l at the beginning of execution. However, the performance characteristics of this scheme are unclear.

10.2.3 Lazy or Eager Synchronization

In the abstract operational semantics, the reduction rules *syncr* and *syncr′* imply a somewhat wasteful model of synchronization. Communication occurs between r' and w every time we synchronize on the value of r' from location w. It is clear that many synchronizations between remote processes could be avoided by memoization. Beyond this obvious optimization, there are some more radical reorganizations of the operational semantics we can consider.

Essentially, the form of the abstract operational semantics was motivated by a desire to delay synchronization as long as possible, permitting the maximum degree of concurrent evaluation. Thus there is no explicit construct forcing synchronization, and we allow spawning dependent processes before their dependencies are fully resolved (new processes may be open with respect to labels r). Lazy synchronization allows more concurrency, but creates problems in resolving addresses and managing communication among a larger number of nodes.

The current ConCert runtime implements a strict synchronization model, in which cords (analogous to processes) are not spawned before all their dependencies are filled with values. All cords which are dependent on a result will be created in an inactive state, waiting for the result to become available. Since dependent cords stay fixed at a known location, it makes sense to assign a *destination* for the result value of each cord. Such a fixed assignment is only tenable because of a prohibition on marshalling or returning values of type α `task`.

By comparing the two evaluation models, we see that the ConCert model is more constrained with respect to concurrent evaluation. However, there are some advantages to these constraints. Strictness allows more efficient execution of purely local code since one does not need runtime checks to distinguish labels from ordinary values. Furthermore, strict spawning allows one to designate a single destination for the result of each cord, simplifying the implementation of synchronization considerably.

10.2.4 Process Garbage Collection

The abstract operational semantics makes no provision for the deletion of a process r, since at any time we may be required to synchronize on r. It is unrealistic to assume that processes persist indefinitely, since a node may fail or choose to leave the grid at any time. Hence we need some means of capturing the result value of such processes and allowing them to disappear upon completion.

We conjecture that it is possible to avoid most of the complexity of distributed garbage collection by reference counting. Because accessibility between processes $w \triangleleft w'$ is generally acyclic, we know that the only form of cycle present in a process configuration involves at least one persistent, globally accessible label l. Since spawning a process might require incrementing counts for a large number of references r, we could use a scheme based on "indirect reference counting" (IRC) to avoid excessive communication. Indirect reference counting is attributed to Piquer [19].

In a sense, garbage collection is a red herring, since we could instead redesign the operational semantics of synchronization to avoid dependencies between arbitrary locations and the obligation to maintain $\langle r' : V \rangle$ indefinitely. For example, we could adopt the point of view that each process r' has a single destination (its parent w), and that all other processes dependent on r' make requests for its value indirectly through parent w. The process r' eagerly forwards its result value (and those of its children) to the parent upon completion. Under this simple scheme, we may delete r' when it both terminates and its number of active children reaches zero.

By using indirection to lookup result values, there is of course the potential to create a bottleneck at some processes in the graph. However, with memoization, the burden could be reduced so that requests for the value bound to r are not duplicated. In any scheme that decouples spawning processes from their scheduling on a concrete node, it seems that some amount of indirection is inevitable. Even in an implementation where two processes can synchronize directly, the parent would be required to serve as a clearinghouse to resolve addresses for its children, since the children could have scheduled in an order not compatible with their dependency structure. Servicing requests for addresses might present less of a bandwidth burden, but the number of requests would be the same.

10.2.5 Implementation of Marshalling

Assume we replace the substitution-based semantics with a classic environment-based one. Let σ denote an environment corresponding to variable context Δ and η denote the analogue for Γ. In the most straightforward presentation, bindings in σ are all of the form $\mathtt{u} \mapsto r$. So we can think of σ as determining some subset of the global environment of processes $\langle r : M \rangle$. The form values for types $A \to B$, $\Box A$, $\Diamond A$, and $\bigcirc A$ will be closures $\{\sigma; \eta; V\}$ for V of the proper form.

Under this kind of semantics, it becomes clear that we will have to implement marshalling for certain forms of closure values as well as the more primitive value types. Marshalling comes into play when we synchronize on the value of a process (*syncr*, *syncr**), spawn a boxed term with $\mathtt{let\ box\,u} = M\ \mathtt{in}\ F$ scheduling it to be run remotely ($letbox_p$ and variants), and jump to the location of some resource using $\mathtt{let\ dia\,x} = M\ \mathtt{in}\ F$ (*syncl*).

Rule	Redex	Marshalled
$letbox_p$	$\langle l : \mathtt{let\ box\,u} = \{\sigma; \eta; \mathtt{box}\, M\}\ \mathtt{in}\ F \rangle$	$\{\sigma; \bullet; M\}$
syncl	$\langle l : \mathtt{let\ dia\,x} = \{\sigma; \eta; \mathtt{dia}\, l'\}\ \mathtt{in}\ F \rangle, \langle l' : \{\sigma'; \eta'; V^*\} \rangle$	$\{\sigma; \mathtt{x} : A \mapsto \bullet; F\}$
syncr*	$\langle r : \{\sigma; \eta; V\} \rangle, \langle l : \mathcal{S}^*[\,r\,] \rangle$	$\{\sigma; \eta; V\}$
syncr*(prim)	$\langle r : V \rangle, \langle l : \mathcal{S}^*[\,r\,] \rangle$	V

When marshalling a closure $\{\sigma; \eta; V\}$, the component V will be some static, compile-time entity $\lambda \mathtt{x} : A \,.\, M$, $\mathtt{box}\, M$, etc. We assume such fragments of code can be compiled to a portable representation, handling marshalling in a static way. But the bindings in σ and η are known only at runtime, and must be marshalled dynamically.

In the case of $letbox_p$, note that the set of local bindings η may be discarded since M is statically known to be Γ-closed. For *syncl* we deviate slightly from the ordinary notion of closures to permit $\{\sigma; \mathtt{x} \mapsto \bullet; F\}$, a special sort of closure with one free local variable $\mathtt{x} : A \mapsto \bullet$. Though $\{\sigma; \mathtt{x} \mapsto \bullet; F\}$ is not closed during transit, it will become closed upon arrival at the destination l'.

Finally, for *syncr* we may be required to marshal the closure $\{\sigma; \eta; V\}$, including local bindings in η. Marshalling of η could pose a problem but for the fact that processes r originate with $\mathtt{box}\, M : \Box A$ and are limited to *pure computations*, hence it should always be possible to crawl over the representation of η at runtime to marshal these values. We note that the static component V could be represented as a code pointer, provided that the receiving location has access to that portion of the compiled program (which will be the case when the receiving location spawned the process r originally).

11 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[4] [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 inherently 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.

Above, we discussed some plans for further investigation into the type theory of mobility by extending the language with polymorphism and abstract types. The plan is to pursue this line of inquiry in order to better characterize immobility in type-theoretic terms. It is hypothesized that type abstraction and perhaps statefulness is the fundamental source of immobility, though perhaps the latter be viewed as an instance of abstraction over the representation of state. From the preliminary discussions above, we see that there are two aspects to this problem. First, we must choose a formalism for polymorphism and/or abstract types, which should be relatively easy since they have been widely studied. Secondly, we will explore various characterizations of marshalling functions, of which the class of strong marshalling functions are but one example. Obviously, mobility, immobility, and our definition of marshalling interact, with weaker notions of marshalling allowing us to claim more types as "mobile". This portion of the investigation should take only a few months; judgment calls will be required as to what notions of weak marshalling are valuable to pursue, since there

[4]The interested reader should see the appendix, section 12.1, 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.

might be an unbounded number of such criteria. Generally, we plan to limit the inquiry by avoiding any application-specific notions of equivalence between abstract datatypes.

As a proof-of-concept, we plan an implementation of a compiler for the calculus generating certified Typed Assembly Language (TAL) [10] code for execution under ConCert [9], a grid infrastructure for distributed programming in development at CMU. ConCert assumes a collection of worker nodes which are organized in a peer-to-peer overlay network. These worker nodes volunteer to execute fragments of closed code (closures) that are statically certified to be safe. It seems clear that the compilation of `box` M and `let dia x =` M `in` F will produce such arbitrarily mobile closures over the code M and F. Though other closure values will be present at runtime, these forms will serve as the basic units of code certification and distribution.

As discussed earlier, there are many issues to be addressed in adapting the abstract specification of the semantics for realistic distributed execution. We plan to defer these issues initially, developing a parser and typechecker for the language first. In this phase, some alterations to the language should be considered to make it more palatable to a programmer, we have in mind ML-style polymorphism and datatypes. As time allows, or in response to pressing needs, we could consider other extensions.

Secondly, we will proceed to develop a simulator back-end for the language. The simulator will allow experimentation with a variety of strategies for scheduling processes and managing synchronization, without the larger engineering effort of generating TAL output or changing the ConCert runtime system. As a first attempt, we plan to implement a fixed-destination semantics similar to the existing ConCert runtime, but permitting a more uniform, higher-order treatment of spawned tasks. The simulator will also be useful for exploring replication and failure-handling strategies, if time allows.

In the final stage of implementation, we plan to implement compilation to TAL, so that programs can be executed in a "trustless" fashion under the ConCert runtime system. The choice of TAL as a target for the compiler is orthogonal to our investigation of the type theory of mobility and locality, but TAL's type safety property is the primary means of providing a static safety guarantee for code distributed to grid participants. We anticipate compilation to TAL will not be significantly more difficult than compilation to untyped assembly language.

We also propose to extend the basic ConCert execution model to support dynamic binding to remote resources. Worker nodes will become distinguishable based on the resources they provide, and the ConCert runtime system will be modified to propagate resource advertisements in addition to work-available notices. Initially, we plan to support some trivial form of remote resources or services as a proof-of-concept, adding additional ones as time allows. We believe the value in achieving real distributed execution under ConCert lies in exploring how the simple, clear programming model of the S4 calculus reduces the required level of runtime support, simplifying marshalling, garbage collection of processes, and perhaps the handling of failures.

References

[1] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanebaum. 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] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. In *GRID 2002 Workshop*, 2002.

[10] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–212. ACM Press, 2003.

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

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

[13] 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.

[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] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *Parallel Architectures and Languages Europe (PARLE)*, volume 505 of *LNCS*, pages 150–165. Springer-Verlag, June 1991.

[20] 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.

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

12 Appendix

12.1 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 $\mathtt{u} :: A$. 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.

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

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 ($J = w\lhd$). We regard $w \lhd \lhd$ as equivalent to $w\lhd$ by definition, so repetitions of $\lhd$ 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 $J = w\lhd$, the relevant judgment holds at all locations accessible from w. Thus logical validity is now associated with the form of judgment $\vdash_{w\lhd}$ in the presence of hypotheses Λ.

We first give the typing rules for labels w. The auxiliary judgment $\psi \vdash^a w \lhd w'$ means that w' is accessible from w under constraints ψ. Constraint entailment $\phi \vdash^a \psi$, presented in section 12.3 of the appendix, defines a small theory of accessibility ($w \lhd w'$) and equivalence ($w \doteq w'$) of locations.

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

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

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:

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

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

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

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

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

Generic effects:

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

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

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

Primitive effects:

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

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

Datatypes:

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

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

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

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

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

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

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.

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

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.

12.2 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 ($\psi \vdash^c C : \Lambda$).

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

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	$\mathcal{R}[\,(\lambda \texttt{x} : A\,.\,M_1')\ \overline{V_2}\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V_2}/\texttt{x}]M_1'\,]$
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	$\mathcal{R}[\,\texttt{case}\ (\texttt{inl}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow N_1 \texttt{ | inr y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V}/\texttt{x}]N_1\,]$
caseR	$\mathcal{R}[\,\texttt{case}\ (\texttt{inr}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow N_1 \texttt{ | inr y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\overline{V}/\texttt{y}]N_2\,]$
$fixpt_v$	$\mathcal{R}[\,\texttt{fixv}\ (\texttt{u} :: A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\![\texttt{fixv}\ (\texttt{u} :: A)\,.\,M/\texttt{u}]\!]M\,]$
fixpt	$\mathcal{R}[\,\texttt{fix}\ (\texttt{x} : A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{R}[\,[\texttt{fix}\ (\texttt{x} : A)\,.\,M/\texttt{x}]M\,]$

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

Rule	E		E'
app*	$\mathcal{S}^*[\,(\lambda \texttt{x} : A\,.\,M_1')\ \overline{V_2}\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V_2}/\texttt{x}]M_1'\,]$
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*	$\mathcal{S}^*[\,\texttt{case}\ (\texttt{inl}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow N_1 \texttt{ | inr y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\texttt{x}]N_1\,]$
caseR*	$\mathcal{S}^*[\,\texttt{case}\ (\texttt{inr}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow N_1 \texttt{ | inr y} \Rightarrow N_2\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\overline{V}/\texttt{y}]N_2\,]$
letdia*	$\texttt{let dia x = dia } E \texttt{ in } F \quad (\text{where } E \neq l')$	$\Longrightarrow$	$\langle\!\langle E/\texttt{x} \rangle\!\rangle F$
$caseL_p$	$\texttt{case}\ (\texttt{inl}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow F_1 \texttt{ | inr y} \Rightarrow F_2$	$\Longrightarrow$	$[\overline{V}/\texttt{x}]F_1$
$caseR_p$	$\texttt{case}\ (\texttt{inr}_{A+B}\ \overline{V})\ \texttt{of inl x} \Rightarrow F_1 \texttt{ | inr y} \Rightarrow F_2$	$\Longrightarrow$	$[\overline{V}/\texttt{y}]F_2$
$fixpt_v^*$	$\mathcal{S}^*[\,\texttt{fixv}\ (\texttt{u} :: A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\![\texttt{fixv}\ (\texttt{u} :: A)\,.\,M/\texttt{u}]\!]M\,]$
fixpt*	$\mathcal{S}^*[\,\texttt{fix}\ (\texttt{x} : A)\,.\,M\,]$	$\Longrightarrow$	$\mathcal{S}^*[\,[\texttt{fix}\ (\texttt{x} : A)\,.\,M/\texttt{x}]M\,]$

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

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

12.3 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.

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

A primitive constraint ($w \triangleleft w'$) 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 $w \triangleleft w'$ or $w' \triangleleft w$ directly. Our notion of accessibility $w \triangleleft w'$ obeys transitivity (from S4) and

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

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

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

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

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

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

12.4 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, $\forall \mathtt{w} \,.\, \mathtt{w} \lhd l$ with the intuitive meaning that l is accessible from anywhere. We also add the dual constraint for labels r, those from which every other location is accessible.

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

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

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

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

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

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

12.5 Properties

Constraints ψ were introduced in section 12.1 and and entailment in 12.3 to describe the allowed dependencies between locations. We say that constraints ψ are *sound* if there are no cycles in accessibility constraints ($\psi \not\vdash^a w \lhd w$).

Definition 3 *A constraint formula ψ is sound (ψ `sound`) iff $\nexists w \,.\, \psi \vdash^a w \lhd w$.*

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:

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

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

Case:

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

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

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

Case:

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

$\Lambda \backslash \psi \vdash^s_l H : \Theta$ — Assumption, Definition
$\Lambda \backslash \psi \vdash^s_{l'} H' : \Theta'$ — Assumption, Definition
$\Theta; \Lambda \backslash \psi; ; \vdash_l \texttt{let dia x = } V \texttt{ in } F \div B$ — Assumption, Definition
$\Theta; \Lambda \backslash \psi; ; \vdash_{l'} \overline{V^*} \div A$ — Assumption, Definition
$; \Lambda \backslash \psi; ; \texttt{x} : A \vdash_{l \triangleleft} F \div B$ — Inversion ($\Diamond E$)
$\Theta; \Lambda \backslash \psi; ; \vdash_l \texttt{dia}\, l' : \Diamond A$ — 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)
$\Theta'; \Lambda' \backslash \psi'; ; \vdash_{l''} \overline{V^*} \div A$ — Weakening, Eq. Worlds ($l' \doteq l''$)
$\Theta'; \Lambda' \backslash \psi'; ; \texttt{x} : A \vdash_{l'' \triangleleft} F \div B$ — Weakening, Natural Mobility
$\Theta'; \Lambda' \backslash \psi'; ; \vdash_{l''} \langle\!\langle \overline{V^*}/\texttt{x}\rangle\!\rangle F \div B$ — Substitution
$\Theta; \Lambda' \backslash \psi'; ; \vdash_l l'' \div B$ — 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.

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

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).