Modal Logic as a Basis for Distributed Computation

Jonathan Moody[1]
jwmoody@cs.cmu.edu

October 2003
CMU-CS-03-194

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Abstract

In this report, we give a computational interpretation of modal logic in which the modalities necessity ($\Box A$) and possibility ($\Diamond A$) describe locality in a distributed computation. This interpretation is quite natural, given the usual "possible worlds" semantics underlying modal logic. In our case, the worlds we consider are processes in a spatially distributed configuration. Necessity describes a term that is well-typed *anywhere* and possibility a term that is well-typed *somewhere*. Thus typing determines the permissible degree of mobility for terms, in some cases allowing us to create new processes or move terms between existing processes, and in others forbidding mobility. In addition to the purely logical motivations, we present some examples demonstrating how the calculus of modal logic proof terms can be used to write distributed, concurrent programs while preserving safe access to and manipulation of localized resources.

Keywords: intuitionistic modal logic, distributed programming, mobility, immobility, Curry-Howard isomorphism

1 Introduction

In this report, we give a computational interpretation of modal logic in which the modalities necessity ($\Box A$) and possibility ($\Diamond A$) describe locality in a distributed computation. This interpretation is quite natural, given the usual "possible worlds" semantics underlying modal logic. In our case, the worlds we consider are processes in a spatially distributed configuration. Necessity describes a term that is well-typed *anywhere* and possibility a term that is well-typed *somewhere*. Thus typing determines the permissible degree of mobility for terms, in some cases allowing us to create new processes or move terms between existing processes, and in others forbidding mobility.

Type	Locality Interpretation
A	type A *here*
$\Box A$	type A *any* (accessible) place
$\Diamond A$	type A *some* (accessible) place

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

2 Modal Logic

Modal logic comes in many varieties; this work is based on an intuitionistic logic of necessity and possibility developed by Pfenning and Davies [13]. This logic resembles S4, in that axioms corresponding to reflexivity and transitivity of accessibility (in the classical setting) are derivable. In later sections of [13], the authors provide a language of proof terms, which can be interpreted as programs via the Curry-Howard isomorphism. We adopt their notation of proof terms for this work as well.

Though Pfenning and Davies gave the outlines of an operational semantics for these proof terms in the form of logically sound local reductions, no particular interpretation of the "worlds" was assumed. Though previous work focused on showing that proof terms of the logic expressed deductions

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

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

in S4 and lax logic (related to monadic programs), this work will show concretely how proof terms express distributed computations. We first extend the notion of a well-formed proof term to a distributed setting in which worlds are reflected concretely as locations (processes) where terms reside. We then give an operational interpretation of such terms in which mobility is logically justified.

Of course, details of the evaluation strategy are not precisely determined given only the logical properties of the language. However, by working from both the logical and the engineering ends of the problem, we show that modal logic proof terms can serve as a sort of calculus for distributed programming. Our results represent one interpretation that we judged best under various practical constraints and desiderata.

2.1 Proof Language

The following term assignment for modal logic is reproduced from [13]. The development of Pfenning and Davies was based on three forms of primitive judgment A `valid`, A `true` and A `poss`, representing the three senses in which we can "know" proposition A holds. Informally these are: A is true in every accessible world (necessity), A is true "here", or A is true in some accessible world (possibility). However, only A `true` and A `poss` are needed to explain the typing rules for the proof language, because A `valid` is defined as deduction of A `true` from no (locally) true assumptions.

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

Two sorts of variable (`x` and `u`) are used to represent hypotheses A `true` and A `valid`, respectively. The distinction between terms and expressions is also logically derived. The expressions are simply those objects which are proofs of A `poss`, whereas terms are those which prove A `true`. The inclusion of terms in the category of expressions (as $\{M\}$) reflects a logical inclusion between truth and possibility. That is, A `true` entails A `poss` in the trivial sense that here is somewhere.

The form of the typing judgment for terms will be $\Delta; \Gamma \vdash M : A$, where Δ and Γ are variable typing contexts corresponding to *valid* and *true* hypotheses, respectively. Implicitly, both hypotheses and conclusion are interpreted as statements about an unspecified current location. The hypotheses in Δ, representing assumptions of A `valid` (here), are available in all accessible worlds. The hypotheses in Γ, corresponding to assumptions of A `true` (here), are only available locally. Since it is not logically sound to permit proofs of A `valid` to depend on local assumptions A `true`, it will be the case that variables $\mathtt{x}$ in Γ have a more restricted scope than $\mathtt{u}$ in Δ. The notation $\mathtt{u} :: A$ will be used to distinguish valid hypotheses from those which are only locally true. Note that the unconventional expression typing judgment $\Delta; \Gamma \vdash E \div A$ is a notation meaning "expression E proves A `poss`".

$$\begin{array}{rrcl}
\text{Types} & A, B & ::= & A \to B \quad | \quad \Box A \quad | \quad \Diamond A \\
\text{Valid Context} & \Delta & ::= & \cdot \quad | \quad \Delta, \mathtt{u} :: A \\
\text{True Context} & \Gamma & ::= & \cdot \quad | \quad \Gamma, \mathtt{x} : A
\end{array}$$

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

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

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

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

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

Note that in rule $\to I$, we treat the new bound variable $\mathtt{x}$ as a "locally true" hypothesis. It will be the case that this non-modal fragment of the logic corresponds to purely local computations expressed in the λ-calculus. The modal fragment, which allows us to make statements about other worlds with $\Box A$ and $\Diamond A$, will allow us to express distributed computations. The typing rules $\Box I$ and $\Diamond E$ deserve special attention, because they impose logically motivated restrictions on hypotheses $\mathtt{x} : A$ of the ordinary, locally true variety.

2.2 Origins of Mobility

Though the language of proof terms and the judgments $\Delta; \Gamma \vdash M : A$ and $\Delta; \Gamma \vdash E \div A$ make no explicit mention of worlds (representing locations), one can gain an intuition for the behavior and mobility of the various terms and expressions through a careful reading of the typing rules. Consider the unusual form of some of the principles of deduction in modal logic, namely $\Box I$ and $\Diamond E$. We will argue that the restrictions they impose on the form of Γ (the locally true hypotheses) provide the logical justification we need to make parts of a program mobile.

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

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

In the case of $\Box I$, reading the rule from the bottom up, if we have a term $\texttt{box}\, M$ proving $\Box A$, we must have a term M of type A, which is closed with respect to Γ and hence well-formed at *any* accessible world. Since M depends on no locally true assumptions in Γ, it makes sense to treat M as being mobile. This observation will permit us to spawn M for evaluation at an arbitrary world. Under the elimination rules $\Box E$ and $\Box E_p$, we see that given $\texttt{box}\, M$ of type $\Box A$, we may rely on the hypothesis $\texttt{u} :: A$ throughout the remainder of the program. Since M establishes A `true` in the absence of local assumptions, we can move M (or its value) to any accessible world, validating the assumption $\texttt{u} :: A$ wherever it occurs. This is the intuition behind the behavior of necessity.

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

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

3 Representing Locality

To this point, we have been speaking abstractly about such things as knowing A `true` in one location and A `poss` in another. We should now develop a notation which reflects such concepts concretely, in the same way that the language of proof terms represents deductions of A `true` or A `poss` relative to an single implicitly defined "current" world. The notation for processes, introduced below, will provide such a mechanism to place proof terms in distinct locations relative to one another.

A single process containing a term in isolation would have no more expressive power than the original calculus of proof terms. It is clear we will need some new form of hypothesis allowing a proof to refer to results established elsewhere (in another process). Process labels are introduced to serve as concrete manifestations of such hypotheses. We distinguish between strong labels (r) corresponding to hypotheses of validity, which we call "result labels" and weak "location labels" (l) corresponding to hypotheses of possibility. Operationally, result labels will allow us to receive the result value of a process, whereas location labels allow us to jump to the location of a remote resource.

$$\text{Process Label} \quad w \quad ::= \quad r \quad | \quad l$$

Processes are labeled by either a result label (r) or location label (l). Labels will serve as process identifiers; we will assume no two processes in a configuration share the same label.

$$\begin{array}{rrcl} \text{Process} & P & ::= & \langle r : M \rangle \quad | \quad \langle l : E \rangle \\ \text{Configuration} & C & ::= & \cdot \quad | \quad C, P \end{array}$$

Process configurations are essentially a labeled collection of terms and expressions. The linear ordering of a process configuration has no special meaning, and we will assume process configurations can be rearranged at will.

Finally, the language of terms is extended to include result labels, and the language of expressions to include location labels.

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

In the context of a proof, a label will serve as a new kind of "remote" hypothesis. We discuss the logical properties of such hypotheses in the following section.

4 Logical Characterization of Processes

Though we defined some notation for process configurations, there is no assurance (as of yet) that such a notation has a well-defined logical meaning. Syntactically a process configuration C is a labeled collection of interdependent proof terms and expressions. We must now provide a definition of well-formedness, which allows us to judge when such a configuration respects the semantics of validity, truth, and possibility in modal logic.

Assuming processes are *closed* with respect to Δ and Γ, that is, $\cdot;\cdot \vdash M : A$ for a process $\langle r : M \rangle$, then it would seem natural to regard the label r as a sort of valid hypothesis, treating it similarly to $\mathtt{u}$. However, there is a subtle distinction to be made between a label r and variable $\mathtt{u}$. In the judgment $\Delta; \Gamma \vdash M : A$, $\mathtt{u} :: A$ denotes the hypothesis that A `valid` is known *here* (implicitly), whereas r refers to a proof of A `valid` located *somewhere else*. In order to remain true to the meaning of A `valid`, we should conclude $\vdash r : A$ at a location only if that location is accessible from r. A similar line of reasoning applies to labels l. Such labels represent the hypothesis that A `poss` is known, not here, but at some other world. To respect the meaning of A `poss`, we should conclude $\vdash l \div A$ at the current location only if l is accessible from our current location. Note that the direction of the required accessibility relationship is reversed when passing between r (logically valid) and l (logically possible) hypotheses.

To accommodate these new kinds of hypotheses in the typing judgment, we introduce a new form of deduction context Λ consisting of a mixed collection of hypotheses $r :: A$ and $l \div A$.

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

The notion that A `valid` known elsewhere can lead to the conclusion A `true` here, and that A `poss` known elsewhere can lead to the conclusion A `poss` here is entirely consistent with the meaning ascribed to these judgments. However, each such case must be justified by some assumption about accessibility between locations (processes). Rather than requiring all such assumptions be mentioned explicitly, it is convenient to represent assumptions about accessibility with a system constraints and entailment on those constraints.[3]

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

[3]It is possible to introduce hypotheses about worlds and accessibility explicitly into the language of proofs, but programs become very rigid in the sense that their layout at runtime is statically determined by typing.

Recall that w denotes a process label r or l. We will treat labels as abstract locations or worlds in a Kripke semantics of modal logic. A primitive constraint $(w \lhd w')$ asserts that accessibility holds between w and w'. The constraint $w \doteq w'$ asserts the equivalence of w and w' under accessibility. That is, both have the same accessibility properties with respect to all other worlds, so in a sense they represent (or share) the "same" location. Compound constraints are conjunctions of such primitive constraints, or the unit element $\top$. When convenient, we may regard a formula ϕ as a set of primitive constraints, joined implicitly by conjunction.

Equivalence $(w \doteq w')$ obeys reflexivity, symmetry, and transitivity, but does *not* entail $w \lhd w'$ or $w' \lhd w$ directly. Accessibility $w \lhd w'$ obeys transitivity (from S4) and respects congruence classes of worlds (as defined by $\doteq$). The S4 assumption of reflexivity $(w \lhd w)$ is not made explicit, but is present in the term and expression typing rules *poss* and *hyp**. The judgment $\phi \vdash^a \psi$, capturing entailment for constraints, is defined as follows. We use Σ to denote a set of formulae $\psi_1, \psi_2, \ldots, \psi_n$.

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

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

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

Note that the specification above is only intended to be complete for derivation of conclusions $w \lhd w'$ and $w \doteq w'$, not arbitrary constraint formulae.

Now if we are to make use of hypotheses in Λ, new forms of hypothetical judgment $\Lambda \backslash \psi; \Delta; \Gamma \vdash_J M : A$ and $\Lambda \backslash \psi; \Delta; \Gamma \vdash_J E \div A$ are needed. These judgments can be understood as a generalization of term and expression typing to a setting in which the relative locations of M (or E) and the hypotheses in Λ are taken into account. The notation $\Lambda \backslash \psi$ is read as Λ subject to ψ, since constraints ψ will determine which hypotheses in Λ are available at a given location. The entire judgment is made relative to a location index J, specifying either a particular location w, or a range of locations, for example $w \lhd$, meaning all locations accessible from w. The relevant forms of index J are:

$$\text{Location Index} \quad J \quad ::= \quad w \quad | \quad J\lhd$$

Though indices J with repetitions of the quantifier $\triangleleft$ are possible ($w \triangleleft \triangleleft \dots$) we consider all such repetitions equivalent to a single one ($w\triangleleft$). That is, $w \triangleleft \triangleleft = w\triangleleft$ by definition. Hence any J is equivalent to one of the canonical forms r, l, $r\triangleleft$, or $l\triangleleft$.

The key rules defining well-formed terms and expressions relative to J are those governing the use of hypotheses in Λ.

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

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

The rules *res* and *loc* are semantically justified by the following observations: If we assume that A `valid` holds in some world r' from which the current location w is accessible, then we can safely conclude A `true` at w. When A `poss` holds in some other world l' accessible from w, the conclusion A `poss` at w is justified. Note that there is no rule corresponding to *ures* for hypotheses l'. Reasoning semantically, we should not assume l' remains available to us at *all* worlds accessible from w.[4] This has the effect of disallowing occurrences of l in the context of the judgment $\vdash_{w\triangleleft}$.

We now proceed to extend the typing judgment to the all other forms of term and expression. These rules do not interact with hypotheses $\Lambda \backslash \psi$, hence we abbreviate $\Lambda \backslash \psi; \Delta; \Gamma \vdash_J M : A$ as $\Delta; \Gamma \vdash_J M : A$ assuming a constant $\Lambda \backslash \psi$ available throughout. The interesting cases are $\Box I$ and $\Diamond E$, where we introduce quantification in the location index ($\vdash_{J\triangleleft}$) for typing

[4]Hypotheses of possibility l can only be permitted under $\vdash_{w\triangleleft}$ if we assume l denotes a globally accessible location ($\forall$`w . w`$\triangleleft l$). We choose not to introduce such assertions of accessibility at this time, since they lead to cycles in accessibility and disrupt the logical reading of process configurations.

certain subterms and subexpressions.

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

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

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

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

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

In the case of $\Box I$ and $\Diamond E$ we require M and F remain well-formed proofs at any world accessible from J ($\vdash_{J\lhd}$). We must do this in the case of $\Box I$, because the boxed proof term M could be required at *all* accessible worlds. For $\Diamond E$, the body of a letbox expression F must be well-formed at the particular location $\mathtt{x}:A$ (a proof of A `true`) is realized. The particular world is unknown to us, hence the requirement that F remain well-formed at *any* accessible world.

By definition, judgments of the form $\vdash_{w\lhd\lhd}$ are equivalent to $\vdash_{w\lhd}$. It is also the case that judgments $\vdash_{w\lhd}$ and $\vdash_w$ are related:

Lemma 4.1 (Typing Inclusion) *If* $\Lambda\backslash\psi;\Delta;\Gamma \vdash_{w\lhd} M:A$ *then* $\Lambda\backslash\psi;\Delta;\Gamma \vdash_w M:A$. *Similarly, if* $\Lambda\backslash\psi;\Delta;\Gamma \vdash_{w\lhd} E\div A$ *then* $\Lambda\backslash\psi;\Delta;\Gamma \vdash_w E\div A$.

Proof: by straightforward induction on typing derivations, making use of the equivalence ($w\lhd\lhd = w\lhd$) when necessary. $\Box$

Given this notion of well-formedness of terms and expressions, we can now define well-formed process configurations. The judgment $\psi \vdash^c C:\Lambda$ means that C establishes Λ under the the assumptions ψ governing accessibility. We define $\psi \vdash^c C:\Lambda$ as follows:

$$\begin{aligned} \psi \vdash^c C:\Lambda \quad \iff & \\ & \mathrm{Dom}(C) = \mathrm{Dom}(\Lambda) \\ & \wedge\ \forall\langle r:M\rangle \in C\ .\ \ \Lambda\backslash\psi;\cdot;\cdot \vdash_{r\lhd} M:\Lambda(r) \\ & \wedge\ \forall\langle l:E\rangle \in C\ .\ \ \Lambda\backslash\psi;\cdot;\cdot \vdash_l E\div\Lambda(l) \end{aligned}$$

The definition 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. Note that ψ determines the "scope" of hypotheses r and l in Λ.

4.1 Accessibility and Soundness

Finally, in light of the role ψ plays in governing the scope of labels, we must reconsider the form of ψ, distinguishing between sound and unsound sets of constraints.

Cyclic constraints $w_0 \lhd w_1 \lhd \ldots \lhd w_0$ can be interpreted as equivalence of $w_0, w_1, \ldots$ in the sense that the labels w_i all share the same accessibility relationships to other locations. However, we consider such cycles *unsound*, since they could permit logically ill-founded process configurations such as $\langle r : r\rangle$ $(\psi = r \lhd r)$ or $\langle l : l'\rangle, \langle l' : l\rangle$ $(\psi = l \lhd l' \wedge l' \lhd l)$. We define soundness of constraints as the absence of cycles in accessibility.

$$\psi \texttt{ csound} \quad\Longleftrightarrow\quad \nexists w \,.\, \psi \vdash_a w \lhd w$$

Explicit equivalence constraints $(w \doteq w')$ are perfectly compatible with this notion of soundness. Here a clear separation between $w \lhd w'$ and $w \doteq w'$ is crucial. The constraint $w \doteq w'$ alone *cannot* permit a cyclic dependency between processes w and w', because the rules for constraint entailment do not define $w \doteq w'$ as $w \lhd w' \wedge w' \lhd w$. By consideration of the typing rules for located hypotheses, we see a true material dependency is only possible if $w \lhd w'$ is known (for equivalence classes of labels w, w'). The intuition is that $w \doteq w'$ equates the locations w and w' of two otherwise *independent* terms or expressions. Our notion of soundness validates this intuition that $w \lhd w'$ and $w \doteq w'$ are mutually exclusive. If there were labels w, w' related by both equivalence $(\psi \vdash^a w \doteq w')$ and accessibility $(\psi \vdash^a w \lhd w')$, then it would also be the case that $\psi \vdash^a w \lhd w$.

Under the requirement ψ `csound` the judgments $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : A$ and $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J E \div A$ become sound with respect to the original notion of well-formed proof.

Theorem 4.1 (Soundness of Process Configuration Typing) *Assume that ψ `csound` and $\psi \vdash^c C : \Lambda$. If $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : A$, then there exists M' such that $\Delta; \Gamma \vdash M' : A$. And if $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J E \div A$, then there exists E' such that $\Delta; \Gamma \vdash E' \div A$.*

Proof: by induction on structure of typing derivation $\vdash_J$ and location index J (ordered by accessibility). Indices J are compared by their root labels, ignoring quantification ($w\lhd = w$). For indices of the form $J\lhd$, we assume the property holds for *prior* J' ($J' \lhd J$). After establishing this, we can proceed to arbitrary J, assuming the property holds for *subsequent* J' ($J \lhd J'$). Both forms of induction hypothesis are sound, because ($\lhd$) is a well-founded strict partial ordering on labels.

Computationally, the proof translates terms and expressions well-typed under $\vdash_J$ by substituting the translation of M from process $\langle r : M\rangle$ for each label r, and E from $\langle l : E\rangle$ for each occurrence of l. This collapses a process configuration into a single term or expression, well-formed under the original $\vdash$ judgment.[5]

Case:

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

$\langle r' : M'\rangle \in C$	Assumption, Definition
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{r'\lhd} M' : A$	Assumption, Definition
$\psi \vdash^a r' \lhd w$	Assumption
There exists N' such that $\cdot; \cdot \vdash N' : A$	IH (accessibility)
$\Delta; \Gamma \vdash N' : A$	Weakening

Case:

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

$\langle r' : M'\rangle \in C$	Assumption, Definition
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{r'\lhd} M' : A$	Assumption, Definition
$\psi \vdash^a r' \lhd w$	Assumption
There exists N' such that $\cdot; \cdot \vdash N' : A$	IH (accessibility)
$\Delta; \Gamma \vdash N' : A$	Weakening

Case:

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

[5] Defining such a substitution operation $\{C/\Lambda\}_\psi M$ explicitly is complicated; a simple simultaneous substitution is not adequate. Rather, we must choose an ordering of Λ (according to certain accessibility criteria) in which M and E are substituted.

$\langle l' : E' \rangle \in C$	Assumption, Definition
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{l'} E' : A$	Assumption, Definition
$\psi \vdash^{a} w \lhd l'$	Assumption
There exists E' such that $\cdot; \cdot \vdash E' \div A$	IH (accessibility)
$\Delta; \Gamma \vdash E' \div A$	Weakening

Case:

$$\frac{\Delta; \cdot \vdash_{J\lhd} M : A}{\Delta; \Gamma \vdash_{J} \texttt{box}\, M : \Box A}\ \Box I$$

$\Lambda \backslash \psi; \Delta; \cdot \vdash_{J\lhd} M : A$	Assumption
There exists M' such that $\Delta; \cdot \vdash M' : A$	IH (derivation)
$\Delta; \Gamma \vdash \texttt{box}\, M' : \Box A$	Typing (rule $\Box I$)

Case:

$$\frac{\Delta; \Gamma \vdash_{J} E \div A}{\Delta; \Gamma \vdash_{J} \texttt{dia}\, E : \Box A}\ \Diamond I$$

$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{J} E \div A$	Assumption
There exists E' such that $\Delta; \Gamma \vdash E' \div A$	IH (derivation)
$\Delta; \Gamma \vdash \texttt{dia}\, E' : \Diamond A$	Typing (rule $\Diamond I$)

Case:

$$\frac{\Delta; \Gamma \vdash_{J} M : A}{\Delta; \Gamma \vdash_{J} \{M\} \div A}\ \mathit{poss}$$

$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{J} M : A$	Assumption
There exists M' such that $\Delta; \Gamma \vdash M' : A$	IH (derivation)
$\Delta; \Gamma \vdash \{M'\} \div A$	Typing (rule *poss*)

Case:

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

$\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : \Diamond A$	Assumption
$\Lambda\backslash\psi; \Delta; \mathtt{x} : A \vdash_{J\triangleleft} F \div B$	Assumption
Exists M' such that $\Delta; \Gamma \vdash M' : \Diamond A$	IH (derivation)
Exists F' such that $\Delta; \mathtt{x} : A \vdash F' \div B$	IH (derivation)
$\Delta; \Gamma \vdash \mathtt{let\ dia\,x} = M' \,\mathtt{in}\, F' \div B$	Typing (rule $\Diamond E$)

□

The judgment $\vdash_J$ is also complete with respect to $\vdash$, in the following sense:

Theorem 4.2 (Completeness of Process Configuration Typing) *If* $\Delta; \Gamma \vdash M : A$ *then* $\cdot\backslash\top; \Delta; \Gamma \vdash_J M : A$ *for any* J. *If* $\Delta; \Gamma \vdash E \div A$ *then* $\cdot\backslash\top; \Delta; \Gamma \vdash_J E \div A$ *for any* J.

Proof: by straightforward induction on derivations $\vdash M : A$ and $\vdash E \div A$. Index J can be chosen arbitrarily because only typing rules for labels (*res*, *loc*, *ures*) constrain the form of J. □

5 An Operational Semantics

In this section, we present a type-sound operational semantics for process configurations. Logical considerations will provide justification of why proofs of a certain form are regarded as mobile whereas others must remain fixed to a certain location. In certain cases, a term (or expression) may be *mobile*, in the sense that $\Lambda\backslash\psi \vdash_J M : A$ and $\Lambda\backslash\psi \vdash_{J'} M : A$ for distinct location indices J and J'. Viewed in this way, the typing judgment expresses the *potential* locations where a term or expression may be placed, not merely its current location. If the operational semantics is to be type-sound, each case in which we move terms or expressions from one world (process) to another must be justified in this way.

We will not assume *a priori* a fixed set of worlds and an accessibility relation constrained by ψ. Rather, it is natural to assume that a proof expression (the program), will reside at a single location initially, but as that program evolves under reduction, certain mobile fragments of the program will be spawned for evaluation in other locations. In each case where such a new process (location) is created, we will assert additional accessibility constraints ψ', essentially defining the new location relative to existing ones.

5.1 Form of Values

Two judgments, M `tvalue` and E `evalue`, define the form of term and expression values, respectively.

$$\frac{}{\lambda\mathtt{x}:A\,.\,M\ \mathtt{tvalue}}\qquad \frac{}{\mathtt{box}\,M\ \mathtt{tvalue}}\qquad \frac{}{\mathtt{dia}\,E\ \mathtt{tvalue}}\qquad \frac{}{r\ \mathtt{tvalue}}$$

$$\frac{V\ \mathtt{tvalue}}{\{V\}\ \mathtt{evalue}}\qquad \frac{}{l\ \mathtt{evalue}}$$

We find it natural to treat the $\Box$ and $\Diamond$ introduction forms ($\mathtt{box}\,M$ and $\mathtt{dia}\,E$) as values, by analogy with the $\rightarrow$ introduction form ($\lambda\mathtt{x} : A\,.\,M$). The result label r is also treated as a value, so that synchronization can be performed lazily. The expression values have the form of either a location label l or a coerced term value $\{V\}$.

We may draw certain conclusions about form of a value given its type. Considering only closed values (Δ and Γ empty), the typing judgment may be abbreviated as $\Lambda\backslash\psi \vdash_J V : A$.

Lemma 5.1 (Typing and Form of Values)

$$\begin{array}{llcll}
V\ \mathtt{tvalue} & \wedge & \Lambda\backslash\psi \vdash_J V : A \rightarrow B & \Longrightarrow & V = \lambda\mathtt{x} : A\,.\,M\ \vee\ V = r \\
V\ \mathtt{tvalue} & \wedge & \Lambda\backslash\psi \vdash_J V : \Box A & \Longrightarrow & V = \mathtt{box}\,M\ \vee\ V = r \\
V\ \mathtt{tvalue} & \wedge & \Lambda\backslash\psi \vdash_J V : \Diamond A & \Longrightarrow & V = \mathtt{dia}\,E\ \vee\ V = r \\
 & & & & \\
V^*\ \mathtt{evalue} & \wedge & \Lambda\backslash\psi \vdash_w V^* \div A & \Longrightarrow & V^* = \{V\}\ \wedge\ V\ \mathtt{tvalue} \\
 & & & & \vee\quad V^* = l
\end{array}$$

Proof: directly, by considering rules defining `tvalue` and `evalue` judgments and rules defining typing judgments. Note that hypothesis rules *res* and *ures* could be used to derive $\vdash_J V : A$ for any type A. Similarly, *loc* can be used to derive $\vdash_w V^* \div A$ for any A. $\Box$

5.2 Definition of Substitution

Pfenning and Davies develop a substitution-based notion of reduction in their paper [13]. Substitution of terms for variables $\mathtt{x}$ ($[M/\mathtt{x}]N$ and $[M/\mathtt{x}]F$) was defined as one would expect, taking into account restrictions on the scope of hypotheses $\mathtt{x} : A$. Substitution of terms for $\mathtt{u} :: A$ ($[\![M/\mathtt{u}]\!]N$ and $[\![M/\mathtt{u}]\!]F$) was also defined in a straightforward way. However, an unusual definition of substitution on expressions was found to be necessary in order to maintain type soundness. Substitution of expressions into expressions (including terms) was defined as follows:

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

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

We have extended the syntax of terms and expressions with labels. Hence it is technically necessary to extend the definition of substitution also. Labels w of both varieties are regarded as insensitive to substitution. The intuition is that labels denote processes which contain terms or expressions that are closed with respect to Δ and Γ.

$$[M/\mathtt{x}]w \;=\; w \qquad [\![M/\mathtt{u}]\!]w \;=\; w$$

$$\langle\!\langle l/\mathtt{x}\rangle\!\rangle F \;=\; \mathtt{let\ dia}\,\mathtt{x}\,\mathtt{=}\,\mathtt{dia}\,l\,\mathtt{in}\,F$$

The case of expression substitution $\langle\!\langle l/\mathtt{x}\rangle\!\rangle F$ is unusual. We cannot simply follow the same strategy used in the prior definition because the form of expression denoted by l is unknown, at least in the context of performing a local substitution. By introducing processes and labels we have created dislocations in terms and expressions, hence reduction cannot always be explained purely by local substitution. A global view of the process configuration as a whole is needed to fully explain the behavior of labels.

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

5.3 Transition Rules

A single-step transition in the semantics is stated as $C \setminus \psi \Longrightarrow C' \setminus \psi'$ for constraints ψ, ψ' and process configurations C, C'. We take the point of view that accessibility constraints 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 of ψ (ψ `csound`) and well-formedness of C with respect to ψ ($\psi \vdash^c C : \Lambda$).

We will be using the notation of evaluation contexts $\mathcal{S}$ to reflect where (in a term or expression) reduction may take place. In fact, evaluation contexts can be split into two definitions, term and expression contexts.

$$
\begin{array}{rrcl}
\text{Term Context} & \mathcal{R} & ::= & [\,] \quad | \quad \mathcal{R}\;M \quad | \quad V\;\mathcal{R} \\
 & & & |\quad \texttt{let box}\,\texttt{u}\,\texttt{=}\,\mathcal{R}\,\texttt{in}\,N \\
\text{Expression Context} & \mathcal{S} & ::= & [\,] \quad | \quad \{R\} \\
 & & & |\quad \texttt{let box}\,\texttt{u}\,\texttt{=}\,\mathcal{R}\,\texttt{in}\,E \\
 & & & |\quad \texttt{let dia}\,\texttt{x}\,\texttt{=}\,\mathcal{R}\,\texttt{in}\,E
\end{array}
$$

Note that only terms M may appear in a context $\mathcal{R}[\,M\,]$. Note also that the structure of $\mathcal{S}$ implies we will only perform reductions on expressions in the empty context ($\mathcal{S} = [\;]$) whereas reductions on terms can occur nested inside of other terms or expressions.

Processes irrelevant to the transition are omitted: $C_1, \langle r : M\rangle, C_2, \langle l : E\rangle, C_3$ is abbreviated as $\langle r : M\rangle, \langle l : E\rangle$. Also recall that the ordering of processes in C is not considered relevant, though an order must be chosen when writing down an instance of that transition.

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

$$
\frac{V_1 = (\lambda \mathtt{x} : A\,.\,M') \quad V_2\ \texttt{tvalue}}{\langle r : \mathcal{R}[\,V_1\ V_2\,]\rangle \setminus \psi \Longrightarrow \langle r : \mathcal{R}[\,[V_2/\mathtt{x}]M'\,]\rangle \setminus \psi}\ \textit{app}
$$

$$
\frac{V_1 = (\lambda \mathtt{x} : A\,.\,M') \quad V_2\ \texttt{tvalue}}{\langle l : \mathcal{S}[\,V_1\ V_2\,]\rangle \setminus \psi \Longrightarrow \langle l : \mathcal{S}[\,[V_2/\mathtt{x}]M'\,]\rangle \setminus \psi}\ \textit{app}'
$$

$$
\frac{V\ \texttt{tvalue}}{\langle r' : V\rangle, \langle r : \mathcal{R}[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : V\rangle, \langle r : \mathcal{R}[\,V\,]\rangle \setminus \psi}\ \textit{syncr}
$$

$$
\frac{V\ \texttt{tvalue}}{\langle r' : V\rangle, \langle l : \mathcal{S}[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : V\rangle, \langle l : \mathcal{S}[\,V\,]\rangle \setminus \psi}\ \textit{syncr}'
$$

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

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

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

$$\frac{\begin{array}{l} V = \mathtt{box}\, M \quad r' \ \mathtt{fresh} \\ \psi' = \psi \wedge (r' \lhd l) \wedge (\bigwedge\{r_i \lhd r' \mid \psi \vdash^a r_i \lhd l\}) \end{array}}{\langle l : \mathtt{let\ box}\, \mathtt{u} \mathtt{=} V \,\mathtt{in}\, F\rangle \setminus \psi \Longrightarrow \langle r' : M\rangle, \langle l : [\![r'/\mathtt{u}]\!] F\rangle \setminus \psi'} \; \mathit{letbox}_p$$

The *letbox* and *letbox′* rules govern the behavior of terms of type $\Box A$. Because the boxed term M is known to be logically valid (and hence mobile) we can spawn an independent process for evaluation of M. Since we are creating an new process r', we must define its relationship to other processes by adding constraints to ψ. Though there are other ways to generate such new constraints, the form of ψ' is intended to be suggestive of creating a new process at a location r' *distinct* from r. The result 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. The rule letbox_p defines the behavior of the variant in which the body F is an expression.

Finally, the *syncl* and *letdia* rules define the behavior of terms $\Diamond A$. Recall that expressions (proofs of A $\mathtt{poss}$) serve as evidence that A is true at some accessible world.

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

In the case of *letdia*, we have direct evidence of A $\mathtt{poss}$ ($E \neq l'$). Therefore E is either $\{M\}$ corresponding to a deduction A $\mathtt{poss}$ because A $\mathtt{true}$ (here), or E is some other form of expression. In either case, we can continue by performing substitution locally. Note that the restriction that E is not a label is crucial because substitution of a label $\langle\!\langle l'/\mathtt{x}\rangle\!\rangle F$ does not allow us to make progress.

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

One can look at *syncl* as a sort of dual of *syncr* – but instead of bringing the immobile expression V^* to our current location, the mobile code F is sent to the location of V^*. Here we have "indirect" evidence l' of A `poss` at some other world. Therefore we jump to that world and resume reduction with the contents of process l'. Note that we must duplicate V^* in $\langle l'' : V^*\rangle$ to preserve the type of the original process l'. The form of the constraints ψ' is intended to suggest creating a process l'' at the *same* location as l', though as with *letbox*, other forms of ψ' are possible. The expression value l'' is produced in the original process to represent the effect of this jump to l''.

5.4 Accessibility Constraints

A few words about the operational interpretation of accessibility constraints are in order. First, note that a single process in isolation, closed with respect to Λ, requires no constraints ($\psi = \top$) in order to be well-formed. Secondly, as the process configuration evolves and additional processes are spawned, the set of constraints will grow monotonically, through the creation of new processes (rule *letbox*) or duplication of processes (rule *syncl*). Thirdly, in interesting initial states $C_0, \langle l_0 : E\rangle \setminus \psi_0$, corresponding to running a program E in an environment C_0, some initial constraints ψ_0 could be required to specify the relative locations of processes in C and the program l_0. Finally, at any given moment, the set of constraints ψ may be stronger than required to ensure well-formedness. Generating or maintaining a minimal ψ requires more detailed program analysis, but would give more precise information about the dependency structure of the program.

There appear to be two ways to view accessibility constraints: either the constraints are informative assertions about dependence between processes (new processes may be placed arbitrarily), or the constraints must be solved at runtime against some *a priori* notion of accessibility (essentially a concrete Kripke model). We have chosen to adopt the former point of view, noting that it is not clear what limitation of the runtime environment a fixed accessibility relation would describe. Accessibility is not precisely communication, since not all communication is conducted in a direction compatible with accessibility.[6] For example, reduction rule *letbox* creates new processes $\langle r' : M\rangle$ by moving M against the direction of accessibility. Accessibility constraints might be useful in other ways when read as assertions about dependency. For example, they might be used to schedule execution and synchronization more efficiently in a lower-level operational semantics.

6 Properties

We will now present type soundness, progress, and confluence theorems for the operational semantics, as well as supporting lemmas. This will demonstrate that the choices we made in defining the operational semantics were correct and logically coherent.

6.1 Substitution

With some generalization, the following substitution properties from [13] hold. As before, a constant $\Lambda\backslash\psi$ deduction context is assumed.

Lemma 6.1 (Properties of Substitution)

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

Proof ($[M/\mathtt{x}]N$ and $[M/\mathtt{x}]F$): by straightforward induction over the typing derivations for N and F, respectively. $\Box$

Proof ($[\![M/\mathtt{u}]\!]N$ and $[\![M/\mathtt{u}]\!]F$): by induction over the typing derivations for N and F, respectively. The specification of a quantified location index $J\triangleleft$ in $\Delta;\cdot \vdash_{J\triangleleft} M : B$ is crucial in the following cases:

Case:

$$\frac{\Delta,\mathtt{u}::B,\Delta';\cdot \vdash_{J\triangleleft} N : A}{\Delta,\mathtt{u}::B,\Delta';\Gamma \vdash_J \mathtt{box}\,N : \Box A}\;\Box I$$

[6]If we also assume symmetry of accessibility, as in the modal logic S5, then viewing accessibility as the capability to communicate might be more tenable.

$\Lambda\backslash\psi; \Delta, \mathtt{u} :: B, \Delta'; \cdot \vdash_{J\triangleleft} N : A$	Assumption
$\Lambda\backslash\psi; \Delta; \cdot \vdash_{J\triangleleft} M : B$	Assumption
$\Lambda\backslash\psi; \Delta; \cdot \vdash_{J\triangleleft\triangleleft} M : B$	Equivalent Index
$[\![M/\mathtt{u}]\!]\mathtt{box}\, N = \mathtt{box}\, [\![M/\mathtt{u}]\!]N$	Definition
$\Lambda\backslash\psi; \Delta, \Delta'; \cdot \vdash_{J\triangleleft} [\![M/\mathtt{u}]\!]N : A$	IH
$\Lambda\backslash\psi; \Delta, \Delta'; \Gamma \vdash_{J} \mathtt{box}\, [\![M/\mathtt{u}]\!]N : \Box A$	Typing (rule $\Box I$)

Case:

$$\frac{\Delta, \mathtt{u} :: B, \Delta'; \Gamma \vdash N : \Diamond C \quad \Delta, \mathtt{u} :: B, \Delta'; \mathtt{x} : C \vdash E \div A}{\Delta, \mathtt{u} :: B, \Delta'; \Gamma \vdash \mathtt{let\ dia}\, \mathtt{x} = N\, \mathtt{in}\, E \div A}\ \Diamond E$$

$\Lambda\backslash\psi; \Delta, \mathtt{u} :: B, \Delta'; \Gamma \vdash_{J} N : \Diamond C$	Assumption
$\Lambda\backslash\psi; \Delta, \mathtt{u} :: B, \Delta'; \mathtt{x} : C \vdash_{J\triangleleft} E \div A$	Assumption
$\Lambda\backslash\psi; \Delta; \cdot \vdash_{J\triangleleft} M : B$	Assumption
$\Lambda\backslash\psi; \Delta; \cdot \vdash_{J\triangleleft\triangleleft} M : B$	Equivalent Index
$[\![M/\mathtt{u}]\!]\mathtt{let\ dia}\, \mathtt{x} = N\, \mathtt{in}\, F = \mathtt{let\ dia}\, \mathtt{x} = [\![M/\mathtt{u}]\!]N\, \mathtt{in}\, [\![M/\mathtt{u}]\!]E$	Definition
$\Lambda\backslash\psi; \Delta, \Delta'; \Gamma \vdash [\![M/\mathtt{u}]\!]N : \Diamond C$	IH
$\Lambda\backslash\psi; \Delta, \Delta'; \mathtt{x} : C \vdash [\![M/\mathtt{u}]\!]E \div A$	IH
$\Lambda\backslash\psi; \Delta, \Delta'; \Gamma \vdash \mathtt{let\ dia}\, \mathtt{x} = [\![M/\mathtt{u}]\!]N\, \mathtt{in}\, [\![M/\mathtt{u}]\!]E \div A$	Typing (rule $\Diamond E$)

$\Box$

Proof ($\langle\!\langle E/\mathtt{x}\rangle\!\rangle F$): by induction over the typing derivations for E, relying on substitution property for $[M/\mathtt{x}]F$.

Case:

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

$\Lambda\backslash\psi; \Delta; \Gamma \vdash_{w} l \div B$	Assumption
$\Lambda\backslash\psi; \Delta; \mathtt{x} : B \vdash_{w\triangleleft} F \div A$	Assumption
$\langle\!\langle l'/\mathtt{x}\rangle\!\rangle F = \mathtt{let\ dia}\, \mathtt{x} = \mathtt{dia}\, l'\, \mathtt{in}\, F$	Definition
$\Lambda\backslash\psi; \Delta; \Gamma \vdash_{w} \mathtt{let\ dia}\, \mathtt{x} = \mathtt{dia}\, l'\, \mathtt{in}\, F \div A$	Typing (rule $\Diamond E$)

Case:

$$\frac{\Delta; \Gamma \vdash_{J} M : B}{\Delta; \Gamma \vdash_{J} \{M\} \div B}\ poss$$

$\Lambda\backslash\psi;\Delta;\Gamma\vdash_J M : B$	Assumption
$\Lambda\backslash\psi;\Delta;\mathtt{x} : B\vdash_{J\lhd} F \div A$	Assumption
$\Lambda\backslash\psi;\Delta;\mathtt{x} : B\vdash_{J} F \div A$	Typing Inclusion
$\Lambda\backslash\psi;\Delta;\Gamma,\mathtt{x} : B\vdash_{J} F \div A$	Weakening
$\langle\!\langle\{M\}/\mathtt{x}\rangle\!\rangle F = [M/\mathtt{x}]F$	Definition
$\Lambda\backslash\psi;\Delta;\Gamma\vdash_J [M/\mathtt{x}]F \div A$	Substitution Prop.

Case:

$$\frac{\Delta;\Gamma\vdash_J M : \Diamond C \quad \Delta;\mathtt{y} : C\vdash_{J\lhd} E \div B}{\Delta;\Gamma\vdash_J \mathtt{let\ dia}\,\mathtt{y}\,\mathtt{=}\,M\,\mathtt{in}\,E \div B}\ \Diamond E$$

$\Lambda\backslash\psi;\Delta;\mathtt{y} : C\vdash_{J\lhd} E \div B$	Assumption
$\Lambda\backslash\psi;\Delta;\Gamma\vdash_J M : \Diamond C$	Assumption
$\Lambda\backslash\psi;\Delta;\mathtt{x} : B\vdash_{J\lhd} F \div A$	Assumption
$\Lambda\backslash\psi;\Delta;\mathtt{x} : B\vdash_{J\lhd\lhd} F \div A$	Equivalent Index
$\langle\!\langle\mathtt{let\ dia}\,\mathtt{y}\,\mathtt{=}\,M\,\mathtt{in}\,E/\mathtt{x}\rangle\!\rangle F = \mathtt{let\ dia}\,\mathtt{y}\,\mathtt{=}\,M\,\mathtt{in}\,\langle\!\langle E/\mathtt{x}\rangle\!\rangle F$	Definition
$\Lambda\backslash\psi;\Delta;\mathtt{y} : C\vdash_{J\lhd} \langle\!\langle E/\mathtt{x}\rangle\!\rangle F \div A$	IH
$\Lambda\backslash\psi;\Delta;\Gamma\vdash_J \mathtt{let\ dia}\,\mathtt{x}\,\mathtt{=}\,M\,\mathtt{in}\,\langle\!\langle E/\mathtt{x}\rangle\!\rangle F \div A$	Typing (rule $\Diamond E$)

Case:

$$\frac{\Delta;\Gamma \vdash_J M : \Box C \quad \Delta, \mathtt{u} :: C;\Gamma \vdash_J E \div B}{\Delta;\Gamma \vdash_J \mathtt{let\ box\,u\,=}\,M\,\mathtt{in}\,E \div B}\ \Box E_p$$

$\Lambda\backslash\psi;\Delta, \mathtt{u} :: C;\Gamma \vdash_J E \div B$	Assumption
$\Lambda\backslash\psi;\Delta;\Gamma \vdash_J M : \Box C$	Assumption
$\Lambda\backslash\psi;\Delta;\mathtt{x} : B \vdash_{J\lhd} F \div A$	Assumption
$\Lambda\backslash\psi;\Delta, \mathtt{u} :: C;\mathtt{x} : B \vdash_{J\lhd} F \div A$	Weakening
$\langle\!\langle \mathtt{let\ box\,u\,=}\,M\,\mathtt{in}\,E/\mathtt{x}\rangle\!\rangle F = \mathtt{let\ box\,u\,=}\,M\,\mathtt{in}\,\langle\!\langle E/\mathtt{x}\rangle\!\rangle F$	Definition
$\Lambda\backslash\psi;\Delta, \mathtt{u} :: C, \Gamma \vdash_J \langle\!\langle E/\mathtt{x}\rangle\!\rangle F \div A$	IH
$\Lambda\backslash\psi;\Delta;\Gamma \vdash_J \mathtt{let\ box\,u\,=}\,M\,\mathtt{in}\,\langle\!\langle E/\mathtt{x}\rangle\!\rangle F \div A$	Typing (rule $\Box E$)

$\Box$

6.2 Mobility

There are a variety of mobility properties which relate the typing judgments $\Lambda\backslash\psi;\Delta;\Gamma \vdash_J M : A$ and $\Lambda\backslash\psi;\Delta;\Gamma \vdash_{J'} M : A$ made relative to distinct locations J and J'. In general, the two judgments are related only if J and J' (when stripped of quantification) are related under accessibility constraints ψ. We analyze various forms of mobility below, noting which reduction rules in the operational semantics make use of each mobility principle.

In the reduction rule *syncr* the following property justifies moving the term value V from w to w'. In the case of *syncl*, it also justifies movement of expression F, the body of a letdia expression. Note that we are moving a term (or expression) typed under the quantified form of typing judgment $\vdash_{w\lhd}$ from w to some accessible location w', a situation which was anticipated when the judgment $\vdash_{w\lhd}$ was defined. Hence this is the simplest, most "natural" form of mobility.

Lemma 6.2 (Natural Mobility ($w \lhd w'$))

$$\begin{array}{lclcl}
\Lambda\backslash\psi;\Delta;\Gamma \vdash_{w\lhd} M : A & \wedge & \psi \vdash^a (w \lhd w') & \Longrightarrow & \Lambda\backslash\psi;\Delta;\Gamma \vdash_{w'\lhd} M : A \\
\Lambda\backslash\psi;\Delta;\Gamma \vdash_{w\lhd} E \div A & \wedge & \psi \vdash^a (w \lhd w') & \Longrightarrow & \Lambda\backslash\psi;\Delta;\Gamma \vdash_{w'\lhd} E \div A
\end{array}$$

Proof: by induction on the typing derivations of M and E. Only the key base case *ures* is shown.

Case:

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

$\psi \vdash^a r' \triangleleft w$	Assumption
$\psi \vdash^a w \triangleleft w'$	Assumption
$\psi \vdash^a r' \triangleleft w'$	Entailment $\vdash^a$ (trans)
$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \triangleleft} r' : A$	Typing (rule *ures*)

$\square$

In reduction rule *syncl* we copy expression value V^* from l' to l''. The intuition is that the duplicate process is be placed at the "same" world, that is $\psi \vdash^a (l' \doteq l'')$. It is always possible to move (in a trivial sense) terms or expressions between equivalent locations.

Lemma 6.3 (Equivalent Worlds ($w \doteq w'$))

$$\begin{array}{rcccl}
\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w \triangleleft} M : A & \wedge & \psi \vdash^a w \doteq w' & \Longrightarrow & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \triangleleft} M : A \\
\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w} M : A & \wedge & \psi \vdash^a w \doteq w' & \Longrightarrow & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w'} M : A \\
\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w \triangleleft} E \div A & \wedge & \psi \vdash^a w \doteq w' & \Longrightarrow & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \triangleleft} E \div A \\
\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w} E \div A & \wedge & \psi \vdash^a w \doteq w' & \Longrightarrow & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w'} E \div A
\end{array}$$

Proof: by induction on the typing derivations of M and E. The key cases are the typing rules for hypotheses r and l.

Case:

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

$\psi \vdash^a r' \triangleleft w$	Assumption
$\psi \vdash^a w \doteq w'$	Assumption
$\psi \vdash^a r' \triangleleft w'$	Entailment $\vdash^a$ (cong.)
$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \triangleleft} r' : A$	Typing (rule *ures*)

Case:

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

$\psi \vdash^a r' \triangleleft w$	Assumption
$\psi \vdash^a w \doteq w'$	Assumption
$\psi \vdash^a r' \triangleleft w'$	Entailment $\vdash^a$ (cong)
$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w'} r' : A$	Typing (rule *res*)

Case:

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

$\psi \vdash^a w \lhd l'$	Assumption
$\psi \vdash^a w \doteq w'$	Assumption
$\psi \vdash^a w' \lhd l'$	Entailment $\vdash^a$ (cong)
$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \lhd} r' : A$	Typing (rule *loc*)

□

In cases when we spawn a new process (*letbox* and variants), we must move a term from w to w' where $w' \lhd w$. Since we cannot assume the term is closed with respect to Λ we must ensure the new location w' is interposed between w and all r_i on which the term might depend. This is the most complex case, because in a sense we are moving against the "natural" direction of accessibility.

Lemma 6.4 (Mobility Against Accessibility ($w' \lhd w$))

$$\begin{array}{ll} & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w \lhd} M : A \\ \wedge & \forall r_i \,.\, (\psi \vdash^a r_i \lhd w) \Rightarrow (\psi \vdash^a r_i \lhd w') \quad \Longrightarrow \quad \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \lhd} M : A \end{array}$$

$$\begin{array}{ll} & \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w \lhd} E \div A \\ \wedge & \forall r_i \,.\, (\psi \vdash^a r_i \lhd w) \Rightarrow (\psi \vdash^a r_i \lhd w') \quad \Longrightarrow \quad \Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \lhd} E \div A \end{array}$$

Proof: by induction on typing derivations for M and E. Only the key base case *ures* is shown.

Case:

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

$\psi \vdash^a r' \lhd w$	Assumption
$\forall r_i \,.\, (\psi \vdash^a r_i \lhd w) \Rightarrow (\psi \vdash^a r_i \lhd w')$	Assumption
Hence $\psi \vdash^a r' \lhd w'$	
$\Lambda \backslash \psi; \Delta; \Gamma \vdash_{w' \lhd} r' : A$	Typing (rule *ures*)

□

6.3 Evaluation Contexts

A key property of evaluation contexts, as they have been defined, is that we never evaluate below a binding construct. Hence we know that the term M' filling the hole in $\mathcal{R}[\,M'\,]$ will be typed in the same combined context $\Lambda\backslash\psi; \Delta; \Gamma$ as the surrounding parts of the term (or expression). For example, if we assume $\mathcal{S}[\,M\,]$ is closed (with respect to Δ and Γ), then M is closed as well.

Lemma 6.5 (Inversion of Typing for Evaluation Contexts) *The following inversion principles apply when typing terms and expressions of the form $\mathcal{R}[\,M\,]$, $\mathcal{S}[\,M\,]$, and $\mathcal{S}[\,E\,]$:*

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

Proof (1): By straightforward induction on the form of $\mathcal{R}$. □

Proof (2): By cases on the form of $\mathcal{S}$, assuming (1) holds for all term evaluation contexts $\mathcal{R}$. □

Proof (3): Since $\mathcal{S}$ could only be $[\,]$, the conclusion is immediate. □

6.4 Type Preservation

The operational semantics is type sound, in the following sense: As the process configuration evolves, new processes may be created, but existing processes remain well-typed (at the same type). The set of accessibility constraints will change to account for the creation of new processes, however, soundness (absence of cycles) of such constraints is preserved.

Theorem 6.1 (Type Preservation) *If ψ `csound`, 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' `csound` and $\psi' \vdash^c C' : \Lambda'$, where Λ' extends Λ.*

$$\begin{array}{rl}
& \psi \;\texttt{csound} \quad\wedge\quad \psi \vdash^c C : \Lambda \quad\wedge\quad C \setminus \psi \Longrightarrow C' \setminus \psi' \\
\Longrightarrow & \exists(\Lambda' \supseteq \Lambda) \;.\; \exists\psi' \;.\quad \psi' \;\texttt{csound} \quad\wedge\quad \psi' \vdash^c C' : \Lambda'
\end{array}$$

Proof: By cases on the $C \setminus \psi \Longrightarrow C' \setminus \psi'$ judgment. Representative cases are shown.

Case:

$$\frac{V\ \texttt{tvalue}}{\langle r' : V\rangle, \langle l : \mathcal{S}[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : V\rangle, \langle l : \mathcal{S}[\,V\,]\rangle \setminus \psi}\ \mathit{syncr}'$$

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

Case:

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

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

Case:

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

$\Lambda\backslash\psi; \cdot; \cdot \vdash_l \texttt{let dia}\, \texttt{x} = V\, \texttt{in}\, F \div B$	Assumption, Definition
$\Lambda\backslash\psi; \cdot; \texttt{x} : A \vdash_{l\triangleleft} F \div B$	Inversion ($\Diamond E$)
$\Lambda\backslash\psi; \cdot; \cdot \vdash_l \texttt{dia}\, E : \Diamond A$	Inversion ($\Diamond E$)
$\Lambda\backslash\psi; \cdot; \cdot \vdash_l E \div A$	Inversion ($\Diamond I$)
$\Lambda\backslash\psi; \cdot; \cdot \vdash_l \langle\!\langle E/\texttt{x}\rangle\!\rangle F \div B$	Substitution
$\psi' = \psi$ and ψ' $\texttt{csound}$	Assumption
$\Lambda' = \Lambda$	Directly

Case:

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

$\Lambda\backslash\psi; \cdot; \cdot \vdash_l \texttt{let dia}\, \texttt{x} = V\, \texttt{in}\, F \div B$	Assumption, Definition
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{l'} V^* \div A$	Assumption, Definition
$\Lambda\backslash\psi; \cdot; \texttt{x} : A \vdash_{l\triangleleft} F \div B$	Inversion ($\Diamond E$)
$\Lambda\backslash\psi; \cdot; \cdot \vdash_l \texttt{dia}\, l' : \Diamond A$	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)
$\Lambda'\backslash\psi'; \cdot; \cdot \vdash_{l''} V^* \div A$	Weakening, Mobility Equivalent Worlds
$\Lambda'\backslash\psi'; \cdot; \texttt{x} : A \vdash_{l''\triangleleft} F \div B$	Weakening, Natural Mobility
$\Lambda'\backslash\psi'; \cdot; \cdot \vdash_{l''} \langle\!\langle V^*/\texttt{x}\rangle\!\rangle F \div B$	Substitution
$\Lambda'\backslash\psi'; \cdot; \cdot \vdash_l l'' \div B$	Typing (*loc*)

l'' $\texttt{fresh}$	Assumption
$\not\exists w, w' \,.\, \psi' \vdash^a w \triangleleft w'$ contradicts ψ $\texttt{csound}$	Form of ψ', Entailment $\vdash^a$
ψ' $\texttt{csound}$	By Contradiction
$\Lambda' \supseteq \Lambda$	Directly

□

6.5 Progress

A progress property for the semantics ensures that well-formed process configurations do not get stuck in an erroneous, non-value, state. The proof of progress relies on the condition ψ `csound`, since the ordering of labels under $w \lhd w'$ must be inductively well-founded.

Theorem 6.2 (Progress) *Assume ψ csound. If $\psi \vdash^c C : \Lambda$, then either C is terminal (all processes contain values) or $C \setminus \psi \Longrightarrow C' \setminus \psi'$ (progress can be made).*

$$\frac{V^* \ \mathit{evalue}}{\langle l : V^* \rangle \ \mathit{terminal}} \qquad \frac{V \ \mathit{tvalue}}{\langle r : V \rangle \ \mathit{terminal}}$$

$$\begin{array}{ll} & \psi \ \mathit{csound} \quad \wedge \quad \psi \vdash^c C : \Lambda \\ \Longrightarrow & C \ \mathit{terminal} \quad \vee \quad \exists (C', \psi') \ . \ C \setminus \psi \Longrightarrow C' \setminus \psi' \end{array}$$

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

$$\begin{array}{ll} & \psi \ \texttt{csound} \ \wedge \ \psi \vdash^c C : \Lambda \ \wedge \ \Lambda \backslash \psi; \cdot; \cdot \vdash_J M : A \\ & \quad \text{(where } J = r\lhd) \\ \Longrightarrow & M \ \texttt{tvalue} \ \vee \ \exists C', M' \ . \ C, \langle r : M \rangle \setminus \psi \Longrightarrow C', \langle r : M' \rangle \setminus \psi' \end{array}$$

$$\begin{array}{ll} & \psi \ \texttt{csound} \ \wedge \ \psi \vdash^c C : \Lambda \ \wedge \ \Lambda \backslash \psi; \cdot; \cdot \vdash_J E \div A \\ & \quad \text{(where } J = l \text{ or } J = l\lhd) \\ \Longrightarrow & E \ \texttt{evalue} \ \vee \ \exists C', E' \ . \ C, \langle l : E \rangle \setminus \psi \Longrightarrow C', \langle l : E' \rangle \setminus \psi' \end{array}$$

The proof then proceeds by induction on the typing derivations for M and E, as well as ordering of location indices J imposed by accessibility constraints ψ. As before, indices J are compared by their root labels w ignoring quantifier symbols. We first consider judgments of the form $J\lhd$, in which case our induction hypothesis is that progress holds for *prior* J' ($J' \lhd J$). Then unquantified J can be considered under the hypothesis that progress holds for *subsequent* J' ($J \lhd J'$). Representative cases are shown:

Case:

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

r' `tvalue` Definition

Case:

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

r' `tvalue` Definition

Case:

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

l' `evalue` Definition

Case:

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

$\lambda \mathtt{x} : A\, .\, M$ `tvalue` Definition

Case:

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

$\Lambda \backslash \psi; \cdot; \cdot \vdash_J M : A \to B$	Assumption
$\Lambda \backslash \psi; \cdot; \cdot \vdash_J N : A$	Assumption
M `tvalue` or $C, \langle r : M \rangle \setminus \psi \Longrightarrow C', \langle r : M' \rangle \setminus \psi'$	IH (derivation)
N `tvalue` or $C, \langle r : N \rangle \setminus \psi \Longrightarrow C', \langle r : N' \rangle \setminus \psi'$	IH (derivation)

Subcase: Progress on either N or M

Progress is also possible for $(M\ N)$ Def. Eval. Context

Subcase: M `tvalue` and N `tvalue`

$M = \lambda\mathtt{x} : A \,.\, M'$ or $M = r'$ — Form of Values

If $M = \lambda\mathtt{x} : A \,.\, M'$ then:
$C, \langle r : M\ N\rangle \setminus \psi \Longrightarrow C, \langle r : [N/\mathtt{x}]M'\rangle \setminus \psi'$ Reduction (rule *app*)

If $M = r'$ then:
$\psi \vdash^{a} r' \lhd r$ — Inversion (*ures*)
Process $\langle r' : M'\rangle \in C$
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{r'\lhd} M' : A \to B$ — Def. Well-formed Conf.
M' `tvalue`
or $C, \langle r : M\ N\rangle, \langle r' : M'\rangle \setminus \psi \Longrightarrow C', \langle r : M\ N\rangle, \langle r' : M''\rangle \setminus \psi'$
IH (accessibility)

In the latter case we are done.
If M' `tvalue` then:
$C, \langle r : r'\ N\rangle \setminus \psi \Longrightarrow C', \langle r : M'\ N\rangle \setminus \psi'$ Reduction (rule *syncr*)

Case:

$$\frac{\Delta; \cdot \vdash_{J\lhd} M : A}{\Delta; \Gamma \vdash_J \mathtt{box}\, M : \Box A}\ \Box I$$

$\mathtt{box}\, M$ `tvalue` — Definition

Case:

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

$\Lambda\backslash\psi; \cdot; \cdot \vdash_J M : \Box A$ — Assumption
$\Lambda\backslash\psi; \mathtt{u} :: A; \cdot \vdash_J N : B$ — Assumption
M `tvalue` or
$C, \langle r : M\rangle \setminus \psi \Longrightarrow C', \langle r : M'\rangle \setminus \psi'$ — IH (derivation)

Subcase: Progress on M.

Progress is also possible for ($\mathtt{let\ box}\, \mathtt{u} = M\, \mathtt{in}\, N$)
Def. Eval. Context

Subcase: M `tvalue`

$M = \texttt{box}\, M'$ or $M = r'$ Form of Values

If $M = \texttt{box}\, M'$ then:
$C, \langle r : \texttt{let box}\, \texttt{u} = M\, \texttt{in}\, N\rangle \setminus \psi \Longrightarrow C, \langle r' : M'\rangle, \langle r : [\![r'/\texttt{u}]\!] N\rangle \setminus \psi'$
Reduction (rule *letbox*)

If $M = r'$ then
$\psi \vdash^a r' \triangleleft r$ Inversion (*ures*)
Process $\langle r' : M'\rangle \in C$
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r'\triangleleft} M' : \Box A$ Def. Well-formed Conf.
M' $\texttt{tvalue}$
or $C, \langle r : ...\rangle, \langle r' : M'\rangle \setminus \psi \Longrightarrow C', \langle r : ...\rangle, \langle r' : M''\rangle \setminus \psi'$
IH (accessibility)

In the latter case we are done.
If M' $\texttt{tvalue}$ then
$C, \langle r : \texttt{let box}\, \texttt{u} = r'\, \texttt{in}\, N\rangle \setminus \psi \Longrightarrow C, \langle r : \texttt{let box}\, \texttt{u} = M'\, \texttt{in}\, N\rangle \setminus \psi'$
Reduction (rule *syncr*)

Case:

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

$\Lambda \backslash \psi; \cdot; \cdot \vdash_J M : \Diamond A$ Assumption
$\Lambda \backslash \psi; \texttt{u} :: A; \cdot \vdash_J F \div B$ Assumption
M $\texttt{tvalue}$ or
$C, \langle r : M\rangle \setminus \psi \Longrightarrow C', \langle r : M'\rangle \setminus \psi'$ IH (derivation)

Subcase: Progress on M.

Progress is also possible for ($\texttt{let dia}\, \texttt{x} = M\, \texttt{in}\, F$)
Def. Eval. Context

Subcase: M $\texttt{tvalue}$

$M = \texttt{dia}\, E$ or $M = r'$ Form of Values

If $M = \texttt{dia}\, E$ and $E \neq l$ then
$C, \langle l : \texttt{let dia}\, \texttt{x} = \texttt{dia}\, E\, \texttt{in}\, F\rangle \setminus \psi \Longrightarrow C, \langle l : \langle\!\langle E/\texttt{x} \rangle\!\rangle F\rangle \setminus \psi'$

Reduction (rule *letdia*)

If $M = \texttt{dia}\, E$ and $E = l$ then

$\psi \vdash^a l \lhd l'$ Inversion (*loc*)

$\langle l' : E' \rangle \in C$

$\Lambda \backslash \psi; \cdot; \cdot \vdash_{l'} E' \div A$ Def. Well-formed Conf.

E' `evalue`

or $C, \langle l : ... \rangle, \langle l' : E' \rangle \setminus \psi \Longrightarrow C', \langle l : ... \rangle, \langle l' : E'' \rangle \setminus \psi'$

IH (accessibility)

In the latter case we are done.

If E' `evalue` then

$C, \langle l : \texttt{let dia}\, \texttt{x} = \texttt{dia}\, l\, \texttt{in}\, F \rangle \setminus \psi$

$\Longrightarrow C, \langle l : l'' \rangle, \langle l' : E' \rangle, \langle l'' : \langle\langle E'/\texttt{x} \rangle\rangle F \rangle \setminus \psi'$ Reduction (rule *syncl*)

If $M = r'$ then

$\psi \vdash^a r' \lhd l$ Inversion (*res*)

Process $\langle r' : M' \rangle \in C$

$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r' \lhd} M' : \Diamond A$ Def. Well-formed Conf.

M' `tvalue`

or $C, \langle l : ... \rangle, \langle r' : M' \rangle \setminus \psi \Longrightarrow C', \langle l : ... \rangle, \langle r' : M'' \rangle \setminus \psi'$

IH (accessibility)

In the latter case we are done.

If M' `tvalue` then:

$C, \langle l : \texttt{let dia}\, \texttt{x} = r'\, \texttt{in}\, N \rangle \setminus \psi \Longrightarrow C, \langle l : \texttt{let dia}\, \texttt{x} = M'\, \texttt{in}\, N \rangle \setminus \psi'$

Reduction (rule *syncr'*)

□

6.6 Termination

Because the basic calculus of proof terms has no primitive fixpoint construct nor are recursive types allowed, it is reasonable to suspect that the operational semantics ($C \setminus \psi \Longrightarrow C' \setminus \psi'$) is terminating. Furthermore, the possibility of cyclic, non-terminating process configurations, such as $\langle r : r \rangle$, is specifically ruled out by the requirement that ψ specify a well-founded accessibility relation. In this section, we establish termination of such well-formed process configurations using the method of logical relations.

Sangiorgi has also applied logical relations successfully in proving termination for a fragment of the Pi calculus [14]. He considers only "functional" processes; in our case, the restriction on accessibility ψ plays a similar role in forcing termination. Though his work encouraged us to believe that logical relations could be applied in the setting of a process calculus, the details of our definitions and proof are quite different.

6.6.1 Definitions

The normal forms under reduction are a subset of what were termed values in prior sections. Though process labels were treated as values in some settings (delaying synchronization), these labels cannot regarded as a proper normal form, since a synchronization rule may apply. We say $C \setminus \psi$ `halts` if all reduction sequences from $C \setminus \psi$ end with a process configuration in normal form, that is, C has no infinite reduction sequences $C \setminus \psi \Longrightarrow C_1 \setminus \psi_1 \Longrightarrow \ldots$. This behavioral criterion defines a subset of configurations for which reduction ($\Longrightarrow$) is strongly normalizing.

In order to reason compositionally about the halting of configurations, we introduce the logical predicates $\mathtt{T}^J_A(M)$ and $\bar{\mathtt{T}}^J_A(E)$ defined on terms $\Lambda \backslash \psi \vdash_J M : A$ and expressions $\Lambda \backslash \psi \vdash_J E \div A$. Note that M or E may be open with respect to process labels in Λ. We also assume the accessibility constraints ψ satisfy ψ `csound`. These predicates characterize a subset of terms/expressions which halt when placed in a process and run in an environment $\psi \vdash^c C : \Lambda$. Of course, M, E, and C must satisfy certain additional conditions.

We now give definitions of $\mathtt{T}^J_A(M)$ and $\bar{\mathtt{T}}^J_A(E)$ which are inductive in J (ordered by accessibility) and type A (structurally). The auxiliary predicates $H(J, M)$ and $H(J, E)$ are introduced as abbreviations. $H(J, M)$ holds if M halts when placed in a process and composed with any terminating configuration C of the proper type. $H(J, E)$ is the analogous condition for expression E.

$$\begin{array}{rcl}
H(J,-) & & \text{(defined for } J \text{ of the form } w \text{ or } w\triangleleft) \\
H(J,M) & \equiv_{\texttt{def}} & \forall C \in \mathbb{T}^{J}_{\Lambda\backslash\psi} \,.\, C, \langle r : M \rangle \setminus \psi \wedge (r \doteq w) \;\texttt{halts} \\
H(J,E) & \equiv_{\texttt{def}} & \forall C \in \mathbb{T}^{J}_{\Lambda\backslash\psi} \,.\, C, \langle l : E \rangle \setminus \psi \wedge (l \doteq w) \;\texttt{halts}
\end{array}$$

$$\begin{array}{rcl}
\mathtt{T}^{J}_{A}(M) & & \text{(defined for } \Lambda\backslash\psi \vdash_J M : A \text{ where } J = r\triangleleft) \\
\mathtt{T}^{J}_{A_0}(M) & \iff & H(J,M) \\
\mathtt{T}^{J}_{A\to B}(M) & \iff & H(J,M) \;\wedge\; \forall N \in \mathtt{T}_{A}.\; \mathtt{T}^{J}_{B}(M\;N) \\
\mathtt{T}^{J}_{\Box A}(M) & \iff & H(J,M) \;\wedge\; \mathtt{T}^{J}_{A}(\texttt{let box}\,\texttt{u}=M\,\texttt{in}\,\texttt{u}) \\
\mathtt{T}^{J}_{\Diamond A}(M) & \iff & H(J,M) \;\wedge\; \bar{\mathtt{T}}^{J}_{A}(\texttt{let dia}\,\texttt{x}=M\,\texttt{in}\,\{\texttt{x}\})
\end{array}$$

$$\begin{array}{rcl}
\mathtt{T}^{J}_{A}(E) & & \text{(defined for } \Lambda\backslash\psi \vdash_J E \div A \text{ where } J = l, J = l\triangleleft) \\
\bar{\mathtt{T}}^{J}_{A_0}(E) & \iff & H(J,E) \\
\bar{\mathtt{T}}^{J}_{A\to B}(E) & \iff & H(J,E) \;\wedge\; \forall N \in \mathtt{T}^{J\triangleleft}_{A} \,.\, \bar{\mathtt{T}}^{J}_{B}(\texttt{let dia}\,\texttt{x}=\texttt{dia}\,E\,\texttt{in}\,\{\texttt{x}\;N\}) \\
\bar{\mathtt{T}}^{J}_{\Box A}(E) & \iff & H(J,E) \;\wedge\; \bar{\mathtt{T}}^{J}_{A}(\texttt{let dia}\,\texttt{x}=\texttt{dia}\,E\,\texttt{in}\,\{\texttt{let box}\,\texttt{u}=\texttt{x}\,\texttt{in}\,\texttt{u}\}) \\
\bar{\mathtt{T}}^{J}_{\Diamond A}(E) & \iff & H(J,E) \;\wedge\; \bar{\mathtt{T}}^{J}_{A}(\texttt{let dia}\,\texttt{x}=\texttt{dia}\,E\,\texttt{in}\,(\texttt{let dia}\,\texttt{y}=\texttt{x}\,\texttt{in}\,\{\texttt{y}\}))
\end{array}$$

Expression termination $\bar{\mathtt{T}}^{J}_{A}(E)$ is clearly related to the corresponding predicate for terms $\mathtt{T}^{J}_{A}$. Indeed, if we consider only the trivial expression $E = \{M\}$ then the criteria for concluding $\bar{\mathtt{T}}^{J}_{A}(\{M\})$ is related to $\mathtt{T}^{J}_{A}(M)$ by a kind of local expansion. But due to the syntactic distinctions between terms and expressions, it is not clear how to combine $\mathtt{T}^{J}_{A}$ and $\bar{\mathtt{T}}^{J}_{A}$ in a single definition.

The predicate $\mathbb{T}^{J}_{\Lambda\backslash\psi}(C)$ characterizes those configurations C consisting solely of processes accessible from index J whose contents satisfy a termination predicate. No extraneous processes that are inaccessible under typing ($\vdash_J$) at judgment index J are permitted. The form of quantification over r, l relative to w is crucial to achieving an inductively well-founded definition. Formally, $\mathbb{T}^{J}_{\Lambda\backslash\psi}(C)$ is defined as:

$$\begin{array}{rcl}
\mathbb{T}^{w}_{\Lambda\backslash\psi}(C) & \iff & \forall r \in \mathrm{Dom}(C) \,.\, \psi \vdash^{a} r \triangleleft w \;\wedge\; \mathtt{T}^{r\triangleleft}_{\Lambda(r)}(C(r)) \\
& & \wedge\; \forall l \in \mathrm{Dom}(C) \,.\psi \vdash^{a} w \triangleleft l \;\wedge\; \bar{\mathtt{T}}^{l}_{\Lambda(l)}(C(l)) \\
\mathbb{T}^{w\triangleleft}_{\Lambda\backslash\psi}(C) & \iff & \forall r \in \mathrm{Dom}(C) \,.\, \psi \vdash^{a} r \triangleleft w \;\wedge\; \mathtt{T}^{r\triangleleft}_{\Lambda(r)}(C(r)) \\
& & \wedge\; \nexists l \in \mathrm{Dom}(C)
\end{array}$$

In the context of a fixed $\Lambda\backslash\psi$ where ψ is sound (acyclic), the predicates $\mathtt{T}^{J}_{A}(M)$ and $\bar{\mathtt{T}}^{J}_{A}(E)$ are inductively well-defined. There are two lexicographic induction orderings, defined on pairs $(w\triangleleft, A)$ and (w, A), respectively. To define the family of termination predicates for $J = w\triangleleft$, each RHS of the definition refers to termination predicates at *prior* labels r or at the same w but with a smaller type. When $J = w$, each RHS refers to $J = w\lhd$ (a family of predicates known to be defined), or a *subsequent* label l, or at the same w with a smaller type.

6.6.2 Global Soundness

We now argue that the termination predicates $\mathtt{T}^J_A(M)$ and $\bar{\mathtt{T}}^J_A(E)$ have the intended meaning, that is, they are sound with respect to halting.

Lemma 6.6 (Global Soundness) *Assume $\psi \vdash^c C : \Lambda$ for ψ `csound`. If all processes r in C satisfy $\mathtt{T}^{r\lhd}_{\Lambda(r)}(C(r))$ and all processes l in C satisfy $\bar{\mathtt{T}}^{l}_{\Lambda(l)}(C(l))$, then $C \setminus \psi$ `halts`.*

Proof: Consider each process $\langle r : M\rangle$ in C. By assumption, we know $\mathtt{T}^{r\lhd}_A(M)$. By definition of the termination predicate, $\forall D \in \mathbb{T}^{r\lhd}_{\Lambda\setminus\psi} . D, \langle r : M\rangle$ `halts`. By the assumption that C is well-formed, and $\Lambda\setminus\psi \vdash_{r\lhd} M : A$, we know that process r is (potentially) dependent on some subset C_r of C, specifically those r' such that $\psi \vdash^a r'\lhd r$. By the assumption that all C satisfy a termination predicate, $\mathbb{T}^{r\lhd}_{\Lambda\setminus\psi}(C_r)$. Hence $C_r, \langle r : M\rangle$ `halts`. The case of a process $\langle l : E\rangle$ is similar, though C_l, the set of (potential) dependencies may consist of both term and expression processes. We conclude that $C_l, \langle l : E\rangle$ `halts`.

For each process, we have a halting fragment $C_r, \langle r : M\rangle$ or $C_l, \langle l : E\rangle$ of the entire configuration C. Note, however, that some of these fragments may overlap and there may be no single fragment encompassing all processes in C.

We argue that C `halts` by contradiction. Assume that C does not halt. Then there exists an infinite reduction sequence S starting from $C \setminus \psi$. For each fragment $C_r, \langle r : M\rangle$ or $C_l, \langle l : E\rangle$, there is a subsequence S_r or S_l of S consisting of reduction steps which apply to that fragment. Due to the way *syncr*, *syncr′* and *syncl* preserve or duplicate processes, each fragment is essentially independent even though some processes may be members of more than one fragment. So each step in the infinite sequence S is present in one or more of the subsequences S_r, S_l. New processes do not arise spontaneously; all new processes are identified with one of the original fragments, which are finite in number. Hence, by a counting argument, at least one of the original fragments supports an infinite reduction sequence. This contradicts the previous result that all such fragments halt. □

6.6.3 Admissibility

In order to show that all well-formed terms and expressions satisfy $\mathtt{T}_A(M)$ or $\bar{\mathtt{T}}_A(E)$, respectively, we must prove certain admissibility/type-closure properties hold for the predicates. Though for the pure lambda calculus, only condition (1) is needed, the calculus of proof terms has a more varied structure requiring further closure conditions. Conditions $(2-4)$ are related to (1) by analogy and allow us to conclude that the various elimination forms are terminating. Conditions $(5-7)$ allow us to conclude that the introduction forms for $\Box A$ and $\Diamond A$, as well as $\{N\}$ are terminating when the term N or expression E is terminating. Conditions $(8-10)$ account for process labels w, which are terminating when the contents of process w is assumed to be terminating.

To prove the lemma by induction on types, the statement of each property must be generalized with an elimination context $\mathcal{E}$. As with evaluation contexts $\mathcal{R}, \mathcal{S}$, elimination contexts come in two varieties $\mathcal{E}, \mathcal{E}'$. $\mathcal{E}[\,M\,]$ denotes a term, and $\mathcal{E}'[\,M\,]$ and $\mathcal{E}'[\,E\,]$ denote expressions.

Lemma 6.7 (Closure/Admissibility Conditions)

$$\begin{array}{llcl}
\textit{Elim. Context} & \mathcal{E} & ::= & [\,]_{\mathtt{term}} \;\mid\; \mathcal{E}\ N \;\mid\; \mathbf{let\ box}\,\mathrm{u} = \mathcal{E}\ \mathbf{in}\,\mathrm{u} \\
 & \mathcal{E}' & ::= & [\,]_{\mathtt{exp}} \;\mid\; \mathbf{let\ dia}\,\mathrm{x} = \mathcal{E}\ \mathbf{in}\,\{\mathrm{x}\} \\
 & & \mid & \mathbf{let\ dia}\,\mathrm{x} = \mathbf{dia}\,\mathcal{E}'\ \mathbf{in}\,\{\mathrm{x}\ N\} \\
 & & \mid & \mathbf{let\ dia}\,\mathrm{x} = \mathbf{dia}\,\mathcal{E}'\ \mathbf{in}\,\{\mathbf{let\ box}\,\mathrm{u} = \mathrm{x}\ \mathbf{in}\,\mathrm{u}\} \\
 & & \mid & \mathbf{let\ dia}\,\mathrm{x} = \mathbf{dia}\,\mathcal{E}'\ \mathbf{in}\,(\mathbf{let\ dia}\,\mathrm{y} = \mathrm{x}\ \mathbf{in}\,\{y\})
\end{array}$$

The following admissibility conditions hold. Due to the $\mathtt{T}^J_A(\mathcal{E}[\,]), \bar{\mathtt{T}}^J_A(\mathcal{E}'[\,])$ *distinction, expression variants exist for (1,2,6,8,9). In these cases, we present only the term variant* $\mathtt{T}^J_A(\mathcal{E}[\,])$.

$$\begin{array}{lrcl}
(1) & \forall N \in \mathtt{T}^J_A\,.\,\mathtt{T}^J_B(\mathcal{E}[\,[N/\mathrm{x}]M\,]) & \Rightarrow & \forall N \in \mathtt{T}^J_A\,.\,\mathtt{T}^J_B(\mathcal{E}[\,(\lambda\mathrm{x}:A\,.\,M)\ N\,]) \\
(2) & \forall N \in \mathtt{T}^{J\lhd}_A\,.\,\mathtt{T}^J_B(\mathcal{E}[\,\llbracket N/\mathrm{u}\rrbracket M\,]) & \Rightarrow & \forall N \in \mathtt{T}^J_{\Box A}\,.\,\mathtt{T}^J_B(\mathcal{E}[\,\mathbf{let\ box}\,\mathrm{u} = N\ \mathbf{in}\,M\,]) \\
(3) & \forall N \in \mathtt{T}^{J\lhd}_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\llbracket N/\mathrm{u}\rrbracket F\,]) & \Rightarrow & \forall N \in \mathtt{T}^J_{\Box A}\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\mathbf{let\ box}\,\mathrm{u} = N\ \mathbf{in}\,F\,]) \\
(4) & \forall E \in \bar{\mathtt{T}}^J_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\langle\!\langle E/\mathrm{x}\rangle\!\rangle F\,]) & \Rightarrow & \forall N \in \mathtt{T}^J_{\Diamond A}\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\mathbf{let\ dia}\,\mathrm{x} = N\ \mathbf{in}\,F\,]) \\
(5) & \forall N \in \mathtt{T}^J_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,[N/\mathrm{x}]F\,]) & \Rightarrow & \forall N \in \mathtt{T}^J_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\mathbf{let\ dia}\,\mathrm{x} = \mathbf{dia}\,\{N\}\ \mathbf{in}\,F\,]) \\
(6) & \forall N \in \mathtt{T}^{J\lhd}_A\,.\,\mathtt{T}^J_B(\mathcal{E}[\,\llbracket N/\mathrm{u}\rrbracket M\,]) & \Rightarrow & \forall N \in \mathtt{T}^{J\lhd}_A\,.\,\mathtt{T}^J_B(\mathcal{E}[\,\mathbf{let\ box}\,\mathrm{u} = \mathbf{box}\,N\ \mathbf{in}\,M\,]) \\
(7) & \forall E \in \bar{\mathtt{T}}^J_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\langle\!\langle E/\mathrm{x}\rangle\!\rangle F\,]) & \Rightarrow & \forall E \in \bar{\mathtt{T}}^J_A\,.\,\bar{\mathtt{T}}^J_B(\mathcal{E}'[\,\mathbf{let\ dia}\,\mathrm{x} = \mathbf{dia}\,E\ \mathbf{in}\,F\,]) \\
(8) & \forall N \in \mathtt{T}^{w\lhd}_A\,.\,\mathtt{T}^{w\lhd}_B(\mathcal{E}[\,N\,]) & \Rightarrow & \forall r :: A \in \Lambda\,.\,\psi \vdash^a r \lhd w \Rightarrow \mathtt{T}^{w\lhd}_B(\mathcal{E}[\,r\,]) \\
(9) & \forall N \in \mathtt{T}^{w}_A\,.\,\mathtt{T}^{w}_B(\mathcal{E}[\,N\,]) & \Rightarrow & \forall r :: A \in \Lambda\,.\,\psi \vdash^a r \lhd w \Rightarrow \mathtt{T}^{w}_B(\mathcal{E}[\,r\,]) \\
(10) & \forall E \in \bar{\mathtt{T}}^{w}_A\,.\,\bar{\mathtt{T}}^{w}_B(\mathcal{E}'[\,E\,]) & \Rightarrow & \forall l \div A \in \Lambda\,.\,\psi \vdash^a w \lhd l \Rightarrow \bar{\mathtt{T}}^{w}_B(\mathcal{E}'[\,l\,])
\end{array}$$

Proof: Each can be proved by induction on type B. In the base case when $B = A_0$, the definitions of $\mathtt{T}^J_A$ and $\bar{\mathtt{T}}^J_A$ are purely behavioral (expressed as the abbreviation $H(J, -)$). By assuming that the compound term in the conclusion does *not* halt, we arrive at a contradiction of the assumptions. The term in the conclusion must halt if we assume the components of that term halt. The same form of argument about the behavior of terms applies at all types, and we omit proofs of $H(J, -)$ in subsequent cases. For the cases $B = A_1 \to A_2$, $B = \Box A_1$, or $B = \Diamond A_1$, we assume the admissibility condition holds for smaller types A_1 and A_2. The definitions of elimination contexts $\mathcal{E}, \mathcal{E}'$ are specifically crafted to allow induction to succeed in these cases.[7]

Case: $B = A_0$ (base type)

Cond: (1)

$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}_{A_0}(\mathcal{E}[\,[N/\mathtt{x}]M\,])$	Assumption
Let: $N \in \mathtt{T}^J_A$	
$H(J, N)$ and $H(J, \mathcal{E}[\,[N/\mathtt{x}]M\,])$	Def. $\mathtt{T}^J_A, \mathtt{T}^J_{A_0}$
(*) Assume not: $H(J, \mathcal{E}[\,(\lambda\mathtt{x} : A \,.\, M)\ N\,])$	
(*) this contradicts $H(J, N)$ or $H(J, \mathcal{E}[\,[N/\mathtt{x}]M\,])$	Def. $\Longrightarrow$
$H(J, \mathcal{E}[\,(\lambda\mathtt{x} : A \,.\, M)\ N\,])$	by Contradiction
$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_{A_0}(\mathcal{E}[\,(\lambda\mathtt{x} : A \,.\, M)\ N\,])$	Def. $\mathtt{T}^J_{A_0}$

Cond: (2-7) similar to (1).

Cond: (8)

$\forall N \in \mathtt{T}^{w\lhd}_A \,.\, \mathtt{T}^{w\lhd}_{A_0}(\mathcal{E}[\,N\,])$	Assumption
Let: $N \in \mathtt{T}^{w\lhd}_A$	
$\forall C \in \mathbb{T}^{w\lhd}_{\Lambda\backslash\psi} \,.\, C, \langle r' : \mathcal{E}[\,N\,]\rangle \setminus \psi \wedge (r' \doteq w)$ `halts`	Def. $\mathtt{T}^{w\lhd}_{A_0}$
$\equiv H(w\lhd, \mathcal{E}[\,N\,])$	
Let: $r :: A \in \Lambda$	
$\psi \vdash^a r \lhd w$	Assumption
Let: $C \in \mathbb{T}^{w\lhd}_{\Lambda\backslash\psi}$	
$\langle r : M\rangle \in C$ and $\mathtt{T}^{r\lhd}_A(M)$	Def. $\mathbb{T}^{w\lhd}_{\Lambda\backslash\psi}$
(*) Assume not: $C, \langle r' : \mathcal{E}[\,r\,]\rangle \setminus \psi \wedge (r' \doteq w)$ `halts`	
(*) This contradicts $H(w\lhd, \mathcal{E}[\,N\,])$ or $\mathtt{T}^{r\lhd}_A(M)$	Def. $\Longrightarrow$

[7]Note that there is a degree of informality in the reasoning about halting of subterms (or subexpressions) $H(J, M)$ (or $H(J, E)$) and halting of compound terms/expressions under $\Longrightarrow$. The relevant lines are marked with (*). This should be clarified in a future revision.

$H(w\lhd, \mathcal{E}[\,r\,])$ by Contradiction
$\forall r :: A \in \Lambda \,.\, \psi \vdash^a r \lhd w \Rightarrow \mathtt{T}^{w\lhd}_{A_0}(\mathcal{E}[\,r\,])$

Cond: (9-10) similar to (8).

Case: $B = \Box A_1$

Cond: (1)

$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_{\Box A_1}(\mathcal{E}[\,[N/\mathrm{x}]M\,])$ Assumption
$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,[N/\mathrm{x}]M\,]\,\mathtt{in\,u})$ Def. $\mathtt{T}^J_{\Box A_1}$
$\forall N \in \mathtt{T}_A \,.\, \mathtt{T}_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,(\lambda \mathrm{x} : A \,.\, M)\ N\,]\,\mathtt{in\,u})$ IH (A_1)
$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_{\Box A_1}(\mathcal{E}[\,(\lambda \mathrm{x} : A \,.\, M)\ N\,])$ Def. $\mathtt{T}^J_{\Box A_1}$

Cond: (2)

$\forall N \in \mathtt{T}^{J\lhd}_A \,.\, \mathtt{T}^J_{\Box A_1}(\mathcal{E}[\,[\![N/\mathrm{u}]\!]M\,])$ Assumption
$\forall N \in \mathtt{T}^{J\lhd}_A \,.\, \mathtt{T}^J_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,[\![N/\mathrm{u}]\!]M\,]\,\mathtt{in\,u})$ Def. $\mathtt{T}^J_{\Box A_1}$
$\forall N \in \mathtt{T}^J_{\Box A} \,.\, \mathtt{T}^J_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,\mathtt{let\ box\,u}\,\mathtt{=}\,N\,\mathtt{in}\,M\,]\,\mathtt{in\,u})$ IH (A_1)
$\forall N \in \mathtt{T}^J_{\Box A} \,.\, \mathtt{T}^J_{\Box A_1}(\mathcal{E}[\,\mathtt{let\ box\,u}\,\mathtt{=}\,N\,\mathtt{in}\,M\,])$ Def. $\mathtt{T}^J_{\Box A_1}$

Cond: (3-7) similar to above.

Cond: (8)

$\forall N \in \mathtt{T}^{w\lhd}_A \,.\, \mathtt{T}^{w\lhd}_{\Box A_1}(\mathcal{E}[\,N\,])$ Assumption
$\forall N \in \mathtt{T}^{w\lhd}_A \,.\, \mathtt{T}^{w\lhd}_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,N\,]\,\mathtt{in\,u})$ Def. $\mathtt{T}^{w\lhd}_{\Box A_1}$
$\forall r :: A \in \Lambda \,.\, \psi \vdash^a r \lhd w \Rightarrow \mathtt{T}^{w\lhd}_{A_1}(\mathtt{let\ box\,u}\,\mathtt{=}\,\mathcal{E}[\,r\,]\,\mathtt{in\,u})$ IH (A_1)
$\forall r :: A \in \Lambda \,.\, \psi \vdash^a r \lhd w \Rightarrow \mathtt{T}^{w\lhd}_{\Box A_1}(\mathcal{E}[\,r\,])$ Def. $\mathtt{T}^{w\lhd}_{\Box A_1}$

Cond: (9-10)

Case: $B = \Diamond A_1$

Cond: (1-6) similar to prior cases.

Cond: (7)

$\forall E \in \bar{\mathtt{T}}^J_A \,.\, \bar{\mathtt{T}}^J_{\Diamond A_1}(\mathcal{E}'[\,\langle\!\langle E/\mathrm{x}\rangle\!\rangle F\,])$ Assumption
Let: $F' = (\mathtt{let\ dia\,y}\,\mathtt{=}\,\mathtt{x\,in\,y})$
$\forall E \in \bar{\mathtt{T}}^J_A \,.\, \bar{\mathtt{T}}^J_{A_1}(\mathtt{let\ dia\,x}\,\mathtt{=}\,\mathtt{dia}\,\mathcal{E}'[\,\langle\!\langle E/\mathrm{x}\rangle\!\rangle F\,]\,\mathtt{in}\,F')$ Def. $\bar{\mathtt{T}}^J_{\Diamond A_1}$
$\forall E \in \bar{\mathtt{T}}^J_A \,.\, \bar{\mathtt{T}}^J_{A_1}(\mathtt{let\ dia\,x}\,\mathtt{=}\,\mathtt{dia}\,\mathcal{E}'[\,\mathtt{let\ dia\,x}\,\mathtt{=}\,\mathtt{dia}\,E\,\mathtt{in}\,F\,]\,\mathtt{in}\,F')$
IH A_1
$\forall E \in \bar{\mathtt{T}}^J_A \,.\, \bar{\mathtt{T}}^J_{\Diamond A_1}(\mathcal{E}'[\,\mathtt{let\ dia\,x}\,\mathtt{=}\,\mathtt{dia}\,E\,\mathtt{in}\,F\,])$ Def. $\bar{\mathtt{T}}^J_{\Diamond A_1}$

Cond: (8-10) similar to prior cases.

Case: $B = A_1 \to A_2$ similar to $B = \Box A_1$ and $B = \Diamond A_1$.

$\Box$

6.6.4 The Fundamental Property

We show that all well-formed terms $(\Delta; \Gamma \vdash_J M : A)$ satisfy $\mathtt{T}^J_A(\sigma M)$ when elements of the substitution σ are assumed to be terminating. An analogous property holds for expressions E. Note that σ satisfies the typing assumptions $\Delta; \Gamma$ and may consist of several forms of substitution – $[\![M/\mathtt{u}]\!]$, $[N/\mathtt{x}]$ or $\langle\!\langle E/\mathtt{y}\rangle\!\rangle$, depending on $\Delta; \Gamma$ and the form of typing judgment. When $\mathtt{T}^{J\lhd}_{\Delta(\mathtt{u})}(M)$, $\mathtt{T}^J_{\Gamma(\mathtt{x})}(N)$, and $\bar{\mathtt{T}}^J_{\Gamma(\mathtt{y})}(E)$, respectively, for all M, N, E components of σ, we write $\mathtt{T}^J_{\Delta;\Gamma}(\sigma)$, meaning σ satisfies the termination conditons for contexts $\Delta; \Gamma$ at J.

Lemma 6.8 (Fundamental Property of Logical Relation) *Assume* $\mathtt{T}^J_{\Delta;\Gamma}(\sigma)$. *That is, σ is a substitution operator satisfying typing assumptions $\Delta; \Gamma$ with terminating bindings. If $\Delta; \Gamma \vdash_J M : A$ then $\mathtt{T}^J_A(\sigma(M))$. And if $\Delta; \Gamma \vdash_w F \div A$ or $\Delta; \mathtt{x_1} : A_1 \vdash_{w\lhd} F \div A$ then $\bar{\mathtt{T}}^w_A(\sigma(F))$. The precise form of σ depends on $\Delta; \Gamma$ and the form of typing judgment $\vdash_J$ as detailed below:*

$$\begin{array}{lll}
\Delta; \Gamma \vdash_w N : A & & \\
\ \wedge\ \sigma = [\![M_1/\mathtt{u_1}]\!] \ldots [\![M_j/\mathtt{u_j}]\!][N_1/\mathtt{x_1}] \ldots [N_k/\mathtt{x_k}] & & \\
\ \wedge\ \forall i\ .\ \mathtt{T}^{w\lhd}_{\Delta(\mathtt{u_i})}(M_i)\ \wedge\ \forall i\ .\ \mathtt{T}^w_{\Gamma(\mathtt{x_i})}(N_i) & \Rightarrow & \mathtt{T}^w_A(\sigma(N)) \\
\Delta; \Gamma \vdash_w F \div A & & \\
\ \wedge\ \sigma = [\![M_1/\mathtt{u_1}]\!] \ldots [\![M_j/\mathtt{u_j}]\!][N_1/\mathtt{x_1}] \ldots [N_k/\mathtt{x_k}] & & \\
\ \wedge\ \forall i\ .\ \mathtt{T}^{w\lhd}_{\Delta(\mathtt{u_i})}(M_i)\ \wedge\ \forall i\ .\ \mathtt{T}^w_{\Gamma(\mathtt{x_i})}(N_i) & \Rightarrow & \bar{\mathtt{T}}^w_A(\sigma(F)) \\
\Delta; \cdot \vdash_{w\lhd} N : A & & \\
\ \wedge\ \sigma = [\![M_1/\mathtt{u_1}]\!] \ldots [\![M_j/\mathtt{u_j}]\!] & & \\
\ \wedge\ \forall i\ .\ \mathtt{T}^{w\lhd}_{\Delta(\mathtt{u_i})}(M_i) & \Rightarrow & \mathtt{T}^{w\lhd}_A(\sigma(N)) \\
\Delta; \mathtt{x_1} : A_1 \vdash_{w\lhd} F \div A & & \\
\ \wedge\ \sigma = [\![M_1/\mathtt{u_1}]\!] \ldots [\![M_j/\mathtt{u_j}]\!]\langle\!\langle E_1/\mathtt{x_1}\rangle\!\rangle & & \\
\ \wedge\ \forall i\ .\ \mathtt{T}^{w\lhd}_{\Delta(\mathtt{u_i})}(M_i)\ \wedge\ \bar{\mathtt{T}}^w_{A_1}(E_1) & \Rightarrow & \bar{\mathtt{T}}^w_A(\sigma(F))
\end{array}$$

Proof: By induction on typing derivations, making extensive use of admissibility conditions (1-10). Some representative cases are presented.

Case:

$$\frac{}{\Delta; \Gamma, \mathtt{x} : A, \Gamma' \vdash_J \mathtt{x} : A}\ \mathit{hyp}$$

$\mathtt{T}^J_{\Delta;\Gamma}(\sigma)$	Assumption
$\mathtt{T}^J_A(\sigma(\mathtt{x}))$	Immediate

Case:

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

$\psi \vdash^a w \lhd l'$	Assumption
$\bar{\mathtt{T}}^w_A(l')$	Admissibility (10)

Case:

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

$\Delta; \Gamma, \mathtt{x} : A \vdash_J M : B$	Assumption
$\mathtt{T}^J_{\Delta;\Gamma}(\sigma)$	Assumption
$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_B(\sigma([N/\mathtt{x}]M))$	IH
$\forall N \in \mathtt{T}^J_A \,.\, \mathtt{T}^J_B(\sigma((\lambda\mathtt{x} : A \,.\, M)\ N))$	Admissibility (1)
$\mathtt{T}^J_{A\to B}(\sigma(\lambda\mathtt{x} : A \,.\, M))$	Def. $\mathtt{T}^J_{A\to B}$

Case:

$$\frac{\Delta; \Gamma \vdash_J M : A}{\Delta; \Gamma \vdash_J \{M\} \div A}\ \mathit{poss}$$

$\Delta; \Gamma \vdash_J M : A$	Assumption
$\mathtt{T}^J_{\Delta;\Gamma}(\sigma)$	Assumption
$\mathtt{T}^J_A(\sigma(M))$	IH
$\bar{\mathtt{T}}^J_A(\sigma(\{M\}))$	Admissibility (5)

Case:

$$\frac{\Delta; \cdot \vdash_{J\lhd} M : A}{\Delta; \Gamma \vdash_J \mathtt{box}\, M : \Box A}\ \Box I$$

$\Delta; \cdot \vdash_{J\triangleleft} M : A$	Assumption
$\mathtt{T}^{J\triangleleft}_{\Delta;\cdot}(\sigma)$	Assumption
$\mathtt{T}^{J\triangleleft}_{A}(\sigma(M))$	IH
$\mathtt{T}^{J}_{\Box A}(\sigma(\mathtt{box}\, M))$	Admissibility (6)

Case:

$$\frac{\Delta; \Gamma \vdash_J E \div A}{\Delta; \Gamma \vdash_J \mathtt{dia}\, E : \Diamond A}\ \Diamond I$$

$\Delta; \Gamma \vdash_J E \div A$	Assumption
$\mathtt{T}^{J}_{\Delta;\Gamma}(\sigma)$	Assumption
$\bar{\mathtt{T}}^{J}_{A}(\sigma(E))$	IH
$\mathtt{T}_{\Diamond A}(\sigma(\mathtt{dia}\, E))$	Admissibility (7)

Case:

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

$\Delta; \Gamma \vdash_J M : A \to B$	Assumption
$\Delta; \Gamma \vdash_J N : A$	Assumption
$\mathtt{T}^{J}_{\Delta;\Gamma}(\sigma)$	Assumption
$\mathtt{T}^{J}_{A\to B}(\sigma(M))$	IH
$\mathtt{T}^{J}_{A}(\sigma(N))$	IH
$\mathtt{T}^{J}_{B}(\sigma(M\ N))$	Def. $\mathtt{T}^{J}_{A\to B}$

Case:

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

$\Delta; \Gamma \vdash_J M : \Box A$	Assumption
$\Delta, \mathtt{u} :: A; \Gamma \vdash_J F \div B$	Assumption
$\mathtt{T}^{J}_{\Delta;\Gamma}(\sigma)$	Assumption
$\mathtt{T}^{J}_{\Box A}(\sigma(M))$	IH
$\forall N \in \mathtt{T}^{J\triangleleft}_{A}\ .\ \bar{\mathtt{T}}^{J}_{B}(\sigma([\![N/\mathtt{u}]\!]F))$	IH
$\forall N \in \mathtt{T}^{J}_{\Box A}\ .\ \bar{\mathtt{T}}^{J}_{B}(\sigma(\mathtt{let\ box}\, \mathtt{u} = N\, \mathtt{in}\, F))$	Admissibility (3)
$\bar{\mathtt{T}}^{J}_{B}(\sigma(\mathtt{let\ box}\, \mathtt{u} = M\, \mathtt{in}\, F))$	Directly

Case:

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

$\Delta; \Gamma \vdash_J M : \Diamond A$	Assumption
$\Delta; \mathtt{x} : A \vdash_{J\lhd} F \div B$	Assumption
$\sigma_1 = [\![M_1/\mathtt{u}_1]\!] \ldots$ and $\sigma_2 = [N_1/\mathtt{x}_1] \ldots$	Assumption
$\mathtt{T}^J_{\Delta;\Gamma}(\sigma_1\sigma_2)$ and $\mathtt{T}^J_{\Delta;\cdot}(\sigma_1)$	Assumption
$\mathtt{T}^J_{\Diamond A}(\sigma_1\sigma_2(M))$	IH
$\forall E \in \bar{\mathtt{T}}^J_A \,.\, \bar{\mathtt{T}}^J_B(\sigma_1(\langle\!\langle E/\mathtt{x}\rangle\!\rangle F))$	IH
$\forall N \in \mathtt{T}^J_{\Diamond A} \,.\, \bar{\mathtt{T}}^J_B(\sigma_1(\mathtt{let\ dia}\,\mathtt{x} = N\,\mathtt{in}\,F))$	Admissibility (4)
$\bar{\mathtt{T}}^J_B(\sigma_1\sigma_2(\mathtt{let\ dia}\,\mathtt{x} = M\,\mathtt{in}\,F))$	Directly

□

Theorem 6.3 (Strong Normalization) *If $\psi \vdash^c C : \Lambda$ then C `halts`.*

Proof: By definition, $\psi \vdash^c C : \Lambda$ implies all processes in C are well-formed. By the fundamental property lemma, processes r satisfy $\mathtt{T}^{r\lhd}_{\Lambda(r)}(C(r))$ and processes l satisfy $\bar{\mathtt{T}}^l_{\Lambda(l)}(C(l))$. By the global soundness lemma, we conclude C `halts`. □

6.7 Confluence

Reduction on configurations $C \setminus \psi \Longrightarrow \psi' \setminus C'$ is nondeterministic. For any configuration C, there may be a choice of process on which to focus, as well as a choice of performing some optional synchronization step(s) (with *syncr* or *syncr*$'$). Though nondeterministic, the operational semantics is confluent modulo a certain notion of equivalence on process configurations C. We will define this equivalence in such a way as to capture precisely the effects of these nondeterministic synchronization steps. Differences in the form of constraints ψ will be ignored, hence $C \setminus \psi \Longrightarrow C' \setminus \psi'$ is abbreviated as $C \Longrightarrow C'$.

Equivalence at the level of terms (and expressions) is defined by the judgment $[M]_C \equiv [N]_D$, meaning that "M (interpreted relative to C) is equivalent to N (relative to D)". There is an implicit side condition that $C \equiv D$, but C and D are not required to be identical. We write simply $M \equiv N$ when the configurations (C, D) are clear from context. Equivalence of expressions is written as $[E]_C \equiv [F]_D$. The $M \equiv N$ relation is simultaneous structural congruence defined by the following axioms and rules (the congruence rules are omitted).

$$\frac{}{[\mathtt{x}]_C \equiv [\mathtt{x}]_D}\;\mathit{eqhyp} \qquad \frac{}{[\mathtt{u}]_C \equiv [\mathtt{u}]_D}\;\mathit{eqhyp}^*$$

$$\frac{}{[r]_C \equiv [r]_D}\;\mathit{eqres} \qquad \frac{}{[l]_C \equiv [l]_D}\;\mathit{eqloc}$$

$$\frac{\langle r : V\rangle \in C \quad V\ \mathtt{tvalue} \quad [V]_C \equiv [V']_D}{[r]_C \equiv [V']_D}\;\mathit{trans}$$

$$\frac{\langle r : V\rangle \in D \quad V\ \mathtt{tvalue} \quad [V']_C \equiv [V]_D}{[V']_C \equiv [r]_D}\;\mathit{trans}'$$

The *trans* and *trans*$'$ rules govern equivalence of labels r and values V' (which may be some other form of term value). The intuition is that synchronization on labels r (rule *syncr* or *syncr*$'$) can be applied at any time. Therefore each label r should be considered interchangeable and equivalent with the corresponding term value in process $\langle r : V\rangle$. On the other hand, location labels l are only equivalent under *eqloc*. We do *not* consider l equivalent to V^* in another process, since rule *syncl* is applied deterministically (within a process) and our goal is to capture precisely the unpredictable aspects of synchronization with equivalence.

Reflexivity is admissible for $(\equiv)$, as are symmetry and transitivity. Reflexivity arises from the structural congruence rules (omitted above) and axioms *eqhyp*, *eqhyp**, etc. The form of *trans* and *trans*$'$ rules were chosen to incorporate symmetry and transitivity.

Lemma 6.9 *Under the definition of $M \equiv N$ (and $E \equiv F$), reflexivity, symmetry, and transitivity of $\equiv$ are admissible.*

Proof: Reflexivity by straightforward induction on the structure of terms and expressions. Symmetry and transitivity by induction on derivations $[M]_C \equiv [N]_D$. $\square$

Equivalence for process configurations ($C \equiv D$) is simply defined as pairwise equivalence of processes. For convenience, we will assume that C and D use identical labels for equivalent processes so that processes are comparable without establishing a mapping between labels of C and those of D.

$$\begin{array}{lcl} C \equiv D & \iff & \mathcal{S} = \mathrm{Dom}(C) = \mathrm{Dom}(D) \\ & & \wedge\ \forall(r \in \mathcal{S})\ .\ [C(r)]_C \equiv [D(r)]_D \\ & & \wedge\ \forall(l \in \mathcal{S})\ .\ [C(l)]_C \equiv [D(l)]_D \end{array}$$

This strong notion of pairwise equivalence is helpful in proving confluence, though an outside observer may only care about equivalence for a distinguished "main" process.

6.7.1 Properties of Equivalence

Derivations of $[M]_C \equiv [N]_D$ are not uniquely invertible, since several rules (namely *eqres*, *trans* and *trans′*), apply to terms of the form r. However, we can identify certain cases based on the form of M and N.

Lemma 6.10 (Inversion of Equivalence) *If $[E]_C \equiv [F]_D$ then corresponding subterms or subexpressions of E and F are equivalent or $E = l = F$. If $[M]_C \equiv [N]_D$ then one of the following holds:*

(1) Neither M nor N is a label (r) and either corresponding subterms or subexpressions of M and N are equivalent (a congruence rule was used) or $M = N$ (rule eqhyp or eqhyp* was used).

(2) $M = r = N$ (rule eqres was used).

(3) $M = r$ and there is a process $\langle r : V\rangle$ in C such that $V \equiv N$ (rule trans). Or $N = r$ and there is a process $\langle r : V\rangle$ in D such that $M \equiv V$ (rule trans′).

Proof: direct, considering cases of $[M]_C \equiv [N]_D$ judgment. □

Equivalence ($\equiv$) is a logical relation in that it relates terms (or expressions) with the same typing properties.

Lemma 6.11 (Typed Equivalence) *Assume $C \equiv D$ where both $\psi \vdash^c C : \Lambda$ and $\psi \vdash D : \Lambda$. Under such Λ and ψ, if $[M]_C \equiv [N]_D$ and $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J M : A$ then $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J N : A$. Also, if $[E]_C \equiv [F]_D$ and $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J E \div A$ then $\Lambda\backslash\psi; \Delta; \Gamma \vdash_J F \div A$.*

Proof: by induction on derivation $[M]_C \equiv [N]_D$ (or $[E]_C \equiv [F]_D$ for expressions). The cases involving labels r are shown:

Case:

$$\frac{\langle r : V\rangle \in C \quad V\ \texttt{tvalue} \quad [V]_C \equiv [V']_D}{[r]_C \equiv [V']_D}\ \mathit{trans}$$

$\psi \vdash^c C : \Lambda$	Assumption
$\langle r : V \rangle \in C$	Assumption
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r \lhd} V : A$	Definition
$[V]_C \equiv [V']_D$	Assumption
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r \lhd} V' : A$	IH

Case:

$$\frac{\langle r : V \rangle \in D \quad V\ \mathtt{tvalue} \quad [V']_C \equiv [V]_D}{[V']_C \equiv [r]_D}\ trans'$$

$\psi \vdash^c D : \Lambda$	Assumption
$\langle r : V \rangle \in D$	Assumption
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r \lhd} V : A$	Definition
$[V']_C \equiv [V]_D$	Assumption
$\Lambda \backslash \psi; \cdot; \cdot \vdash_{r \lhd} V' : A$	Symmetry, IH

□

Equivalence is compatible with the definition of term and expression values, in the sense that values are equivalent to other values.

Lemma 6.12 (Equivalence of Values) *If M tvalue and $[M]_C \equiv [N]_D$ then N tvalue. If E evalue and $[E]_C \equiv [F]_D$ then F tvalue.*

Proof: For term values, the proof is by induction on derivations of $M \equiv N$, considering cases consistent with M `tvalue`. The analogous proof for expression values is also straightforward, and relies on the property we just established for equivalence of term values. □

Equivalence is compatible with substitution in the sense that substitution applied to equivalent terms or expressions yields equivalent results. Note that all terms, expressions, and process configurations are assumed to be well-formed.

Lemma 6.13 (Equivalence Compatible with Substitution) *If C and C' are well-formed and $C \equiv C'$ then the following hold:*

$$\begin{array}{rcrcl}
[M]_C \equiv [M']_{C'} & \wedge & [N]_C \equiv [N']_{C'} & \Longrightarrow & [[M/\mathtt{x}]N]_C \equiv [[M'/\mathtt{x}]N']_{C'} \\
[M]_C \equiv [M']_{C'} & \wedge & [F]_C \equiv [F']_{C'} & \Longrightarrow & [[M/\mathtt{x}]F]_C \equiv [[M'/\mathtt{x}]F']_{C'} \\
[M]_C \equiv [M']_{C'} & \wedge & [N]_C \equiv [N']_{C'} & \Longrightarrow & [[\![M/\mathtt{u}]\!]N]_C \equiv [[\![M'/\mathtt{u}]\!]N']_{C'} \\
[M]_C \equiv [M']_{C'} & \wedge & [F]_C \equiv [F']_{C'} & \Longrightarrow & [[\![M/\mathtt{u}]\!]F]_C \equiv [[\![M'/\mathtt{u}]\!]F']_{C'} \\
[E]_C \equiv [E']_{C'} & \wedge & [F]_C \equiv [F']_{C'} & \Longrightarrow & [\langle\!\langle E/\mathtt{x}\rangle\!\rangle F]_C \equiv [\langle\!\langle E'/\mathtt{x}\rangle\!\rangle F']_{C'}
\end{array}$$

Proof ($[M/\mathtt{x}]N$ and $[M/\mathtt{x}]F$): by induction on the derivation $[N]_C \equiv [N']_{C'}$ (or $[F]_C \equiv [F']_{C'}$). Some representative base cases are shown:

Case:

$$\frac{}{[r]_C \equiv [r]_{C'}}\ \mathit{eqres}$$

$[M/\mathtt{x}]r = r$ and $[M'/\mathtt{x}]r = r$	Definition
$[[M/\mathtt{x}]r]_C \equiv [[M'/\mathtt{x}]r]_{C'}$	Equivalence (rule *eqres*)

Case:

$$\frac{}{[l]_C \equiv [l]_{C'}}\ \mathit{eqloc}$$

$[M/\mathtt{x}]l = l$ and $[M'/\mathtt{x}]l = l$	Definition
$[[M/\mathtt{x}]l]_C \equiv [[M'/\mathtt{x}]l]_{C'}$	Equivalence (rule *eqloc*)

Case:

$$\frac{\langle r : V\rangle \in C \quad V\ \mathtt{tvalue} \quad [V]_C \equiv [V']_{C'}}{[r]_C \equiv [V']_{C'}}\ \mathit{trans}$$

$[M/\mathtt{x}]r = r$	Definition
$\langle r : V\rangle \in C$ and V `tvalue` and $[V]_C \equiv [V']_{C'}$	Assumption
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{r\lhd} V : A$	Def. Well-formed Conf.
$\Lambda\backslash\psi; \cdot; \cdot \vdash_{r\lhd} V' : A$	Equiv. Typed
$[M'/\mathtt{x}]V' = V'$	Subst. on Closed Term
$[[M/\mathtt{x}]r]_C \equiv [[M'/\mathtt{x}]V']_{C'}$	Equivalence (rule *trans*)

□

Proof ($[\![M/\mathtt{u}]\!]N$ and $[\![M/\mathtt{u}]\!]F$): by induction on the derivation $[N]_C \equiv [N']_{C'}$ (or $[F]_C \equiv [F']_{C'}$). The proof is straightforward and quite similar to the case of ordinary substitution ($[M/\mathtt{x}]N$ and $[M/\mathtt{x}]F$). □

Proof ($\langle\!\langle E/\mathtt{x}\rangle\!\rangle F$): by induction on the derivation $[E]_C \equiv [E']_{C'}$, making use of the equivalence result for term substitution established above. A few representative cases are show:

Case:

$$\frac{}{[l]_C \equiv [l]_{C'}}\ \mathit{eqloc}$$

$\langle\langle l/\mathtt{x}\rangle\rangle F = \mathtt{let\ dia\,x = dia}\,l\,\mathtt{in}\,F$
and $\langle\langle l/\mathtt{x}\rangle\rangle F' = \mathtt{let\ dia\,x = dia}\,l\,\mathtt{in}\,F'$ Definition
$[l]_C \equiv [l]_{C'}$ Assumption
$[F]_C \equiv [F']_{C'}$ Assumption
$[\mathtt{dia}\,l]_C \equiv [\mathtt{dia}\,l]_{C'}$ Equivalence (cong. rule)
$[\langle\langle l/\mathtt{x}\rangle\rangle F]_C \equiv [\langle\langle l/\mathtt{x}\rangle\rangle F']_{C'}$ Equivalence (cong. rule)

Case:

$$\frac{[M]_C \equiv [M']_{C'}}{[\{M\}]_C \equiv [\{M'\}]_{C'}}\ \mathit{eqposs}$$

$\langle\langle \{M\}/\mathtt{x}\rangle\rangle F = [M/\mathtt{x}]F$ Definition
$\langle\langle \{M'\}/\mathtt{x}\rangle\rangle F' = [M'/\mathtt{x}]F'$ Definition
$[M]_C \equiv [M']_{C'}$ and $[F]_C \equiv [F']_{C'}$ Assumption
$[[M/\mathtt{x}]F]_C \equiv [[M'/\mathtt{x}]F']_{C'}$ Compatibility with Subst.
$[\langle\langle \{M\}/\mathtt{x}\rangle\rangle F]_C \equiv [\langle\langle \{M'\}/\mathtt{x}\rangle\rangle F']_{C'}$ Definition

Case:

$$\frac{[M]_C \equiv [M']_{C'} \quad [E]_C \equiv [E']_{C'}}{[\mathtt{let\ dia\,x = }M\,\mathtt{in}\,E]_C \equiv [\mathtt{let\ dia\,x = }M\,\mathtt{in}\,E]_{C'}}\ \mathit{eq}\Diamond E$$

$\langle\langle \mathtt{let\ dia\,x = }M\,\mathtt{in}\,E/\mathtt{x}\rangle\rangle F = \mathtt{let\ dia\,x = }M\,\mathtt{in}\,\langle\langle E/\mathtt{x}\rangle\rangle F$ Definition
$[F]_C \equiv [F']_{C'}$ Assumption
$[M]_C \equiv [M']_{C'}$ and $[E]_C \equiv [E']_{C'}$ Assumption
$[\langle\langle E/\mathtt{x}\rangle\rangle F]_C \equiv [\langle\langle E'/\mathtt{x}\rangle\rangle F']_{C'}$ IH (derivation)
$[\langle\langle \mathtt{let\ dia\,x = }M\,\mathtt{in}\,E/\mathtt{x}\rangle\rangle F]_C \equiv [\langle\langle \mathtt{let\ dia\,x = }M\,\mathtt{in}\,E/\mathtt{x}\rangle\rangle F']_{C'}$
Equivalence (cong. rule)

□

Equivalence is also compatible with the formation of evaluation contexts, in the sense that decompositions $\mathcal{R}[\,M'\,]$ are related to "equivalent" decompositions $\mathcal{R}'[\,N'\,]$.

Lemma 6.14 *If $M \equiv N$ and $N = \mathcal{R}[\,N'\,]$ then there exists $\mathcal{R}'$ and M' such that $M = \mathcal{R}'[\,M'\,]$ and $M' \equiv N'$. If $E \equiv F$ and $F = \mathcal{S}[\,N'\,]$ then there exists $\mathcal{S}'$ and M' such that $E = \mathcal{S}'[\,M'\,]$ and $M' \equiv N'$.*

Proof: By induction on the structure of evaluation contexts. We note that only case (1) of the equivalence inversion lemma applies when either M or N is *not* a value. A representative case is shown:

Case: $\mathcal{R}[\,N'\,] = V_1\ \mathcal{R}'[\,N'\,]$

$M \equiv V_1\ \mathcal{R}'[\,N'\,]$	Assumption
$M = V_1'\ M_2$ and $V_1' \equiv V_1$ and $M_2 \equiv \mathcal{R}'[\,N'\,]$	Inversion
There exists $\mathcal{R}''[\ ]$ such that $M_2 = \mathcal{R}''[\,M'\,]$ and $M' \equiv N'$	IH
$M\ =\ V_1'\ \mathcal{R}''[\,M'\,]\ =\ \mathcal{R}'''[\,M'\,]$ and $M' \equiv N'$	Def. of Ev. Context

□

6.7.2 Equivalence and Reduction

We can now proceed to analyze the interaction between equivalence and reduction in certain restricted cases. A number of lemmas are proved which will be of use later in establishing the confluence result.

The first of these is that equivalence ($C \equiv D$) does, in fact, capture the synchronization steps which we wish to ignore.

Lemma 6.15 (Synchronization Preserves Equivalence Class) *For well-formed configurations C, if $C \Longrightarrow D$ is made via the rule syncr or syncr$'$, then $C \equiv D$.*

Note that the converse of this property does not hold in general, because reduction can only occur in certain contexts $\mathcal{S}[\ ]$ or $\mathcal{R}[\ ]$, not in arbitrary locations of the term or expression. Thus equivalence does not imply *convertibility* of terms in one direction or the other.

Proof: direct, considering the two reduction rules *syncr* and *syncr$'$*. The case of *syncr$'$* is shown:

Case:

$$\frac{V\ \texttt{tvalue}}{\langle r' : V\rangle, \langle l : \mathcal{S}[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : V\rangle, \langle l : \mathcal{S}[\,V\,]\rangle \setminus \psi}\ \mathit{syncr}'$$

$[V]_C \equiv [V]_D$	Reflexivity
$[r']_C \equiv [V]_D$	Equivalence (rule *trans*)
$[\mathcal{S}[\,r'\,]]_C \equiv [\mathcal{S}[\,V\,]]_D$	Equivalence (cong. rule(s))
$C \equiv D$	Definition

□

Though equivalent terms M and N are not always convertible to syntactically equal forms, if we restrict our attention to values, it is clear that we can perform a series of synchronization steps to reach observationally equivalent terms.[8]

Observational equivalence $[M]_C \equiv_o [N]_D$ is defined on term and expression values. It is stronger than general equivalence, that is, $M \equiv_o N$ implies $M \equiv N$. Essentially, $M \equiv_o N$ requires that $M \equiv N$ *and* both M and N have the same top-level form.

$$\frac{}{[r]_C \equiv_o [r]_D} \qquad \frac{[M]_C \equiv [N]_D}{[\lambda \mathtt{x} : A \,.\, M]_C \equiv_o [\lambda \mathtt{x} : A \,.\, N]_D} \qquad \frac{[M]_C \equiv [N]_D}{[\mathtt{box}\, M]_C \equiv_o [\mathtt{box}\, N]_D}$$

$$\frac{}{[l]_C \equiv_o [l]_D} \qquad \frac{[V]_C \equiv_o [V]_D}{[\{V\}]_C \equiv_o [\{V\}]_D} \qquad \frac{[E]_C \equiv [F]_D}{[\mathtt{dia}\, E]_C \equiv_o [\mathtt{dia}\, F]_D}$$

Lemma 6.16 (Equivalence of Values implies Weak Convertibility) *If M **tvalue** and $[M]_C \equiv [N]_D$ then M and N are convertible to observationally equivalent forms $[M']_C \equiv_o [N']_D$ via reduction sequences $C, \langle r : M \rangle \Longrightarrow^* C, \langle r : M' \rangle$ and $D, \langle r : N \rangle \Longrightarrow^* D, \langle r : N' \rangle$. For expressions, if E **evalue** and $[E]_C \equiv [F]_D$ then E and F are convertible to observationally equivalent forms $[E']_C \equiv_o [F']_D$ via $C, \langle l : E \rangle \Longrightarrow^* C, \langle l : E' \rangle$ and $D, \langle l : F \rangle \Longrightarrow^* D, \langle l : F' \rangle$*

Proof ($M \equiv N$): by induction on the derivation $[M]_C \equiv [N]_D$, considering cases which are compatible with M `tvalue` and N `tvalue`. Only the cases corresponding to *trans* and *trans'* involve non-trivial reduction sequences.

Case:

$$\frac{}{[r]_C \equiv [r]_D}\ \textit{eqres}$$

$r \equiv_o r$ (no reduction steps are required) Definition

Case:

$$\frac{\langle r : V \rangle \in C \quad V\ \mathtt{tvalue} \quad [V]_C \equiv [V']_D}{[r]_C \equiv [V']_D}\ \textit{trans}$$

[8]The restriction to values limits the scope of this lemma, making the proof manageable. We will later show confluence holds for arbitrary terms.

$\langle r : V\rangle \in C$ and V `tvalue`	Assumption
$C, \langle r' : r\rangle \Longrightarrow C, \langle r' : V\rangle$	Reduction (rule *syncr*)
$[V]_C \equiv [V']_D$	Assumption
V' `tvalue`	Assumption

$C, \langle r' : V\rangle \Longrightarrow^* C, \langle r' : M'\rangle$
and $D, \langle r' : V'\rangle \Longrightarrow^* D, \langle r' : N'\rangle$
such that $M' \equiv_o N'$ IH

□

Proof ($E \equiv F$): by cases on the derivation $[E]_C \equiv [F]_D$, assuming the property holds for all term values. □

Within a process $\langle r : M\rangle$ or $\langle l : E\rangle$, the decomposition of M into $\mathcal{R}[\,M'\,]$ (or E into $\mathcal{S}[\,M\,]$ or $\mathcal{S}[\,E'\,]$) is not uniquely determined (in a strict sense). Typically, values are not allowed to be redices, but our semantics makes an exception for result labels, $\mathcal{R}[\,r'\,]$ via the *syncr* rule (and $\mathcal{S}[\,r'\,]$ via *syncr'*). This exception leads to many possible decompositions of a term, and hence many possible reduction steps within a single process. We will show that all of these choices (except one) correspond to optional synchronization steps. We note that redices have the forms: $((\lambda \mathtt{x} : A \,.\, M')\ V_2)$, $(\mathtt{let\ box\,u = box}\,M\,\mathtt{in}\,N)$, $(\mathtt{let\ box\,u = box}\,M\,\mathtt{in}\,F)$, $(\mathtt{let\ dia\,x = dia}\,E\,\mathtt{in}\,F)$, or (r').

Lemma 6.17 (Unique Evaluation Contexts (excluding redices r)) *Any well-formed term M (or expression E) is either a value or has at most one decomposition as $\mathcal{R}[\,M'\,]$ (or $\mathcal{S}[\,M'\,]$, $\mathcal{S}[\,E'\,]$) where M' and E' are redices and $M' \neq r$. If redices $M' = r$ are also considered, then there will be one or more such decompositions.*

Proof: by a straightforward induction on the structure of terms and expressions. Only the form of function application ($M\ N$) allows more than one decomposition (when redices r' are considered). We summarize the possibilities for decomposing $(M\ N)$ in the table below:

	Form of $(M\ N)$	Form of V	Reduction(s)	Context Extension(s)
$(1a)$	$(\lambda \mathtt{x} : A \,.\, M')\ V$	$V = r$	*app*	$\mathcal{R}[\,\lambda \mathtt{x} : A \,.\, M'\ V\,] \rightarrow \mathcal{R}'[\,V\,]$
$(1b)$		$V \neq r$	*app*	
$(2a)$	$V\ N$	$V = r$		$\mathcal{R}[\,V\ N\,] \rightarrow \mathcal{R}'[\,V\,]$
				$\mathcal{R}[\,V\ N\,] \rightarrow \mathcal{R}'[\,N\,]$
$(2b)$		$V = \lambda \mathtt{x} : A \,.\, M'$		$\mathcal{R}[\,V\ N\,] \rightarrow \mathcal{R}'[\,N\,]$
(3)	$M\ N$	(none)		$\mathcal{R}[\,M\ N\,] \rightarrow \mathcal{R}'[\,M\,]$

Metavariable V denotes a term value. Cases $(1a)$ and $(2a)$ are the source of nondeterminism. In $(1a)$, we can treat $((\lambda \mathtt{x} : A \,.\, M)\ r)$ as a redex, applying rule *app*, or we can further decompose this term as $\mathcal{R}'[\,r\,]$, synchronizing on r. In $(2a)$, we can decompose the term two ways, focusing either on the function position $(\mathcal{R}'[\,r\,])$ or the argument position $(\mathcal{R}'[\,N\,])$. If we disallow decompositions $\mathcal{R}'[\,r\,]$ then this nondeterminism in cases $(1a)$ and $(2a)$ disappears, and at most one decomposition of $(M\ N)$ possible. The term $(r\ V)$ will be the critical case in which no decomposition exists, synchronization being mandatory in such cases. Now if we also permit decompositions $\mathcal{R}'[\,r\,]$, there will be one or more such decompositions. All but one of these decompositions will correspond to optional synchronizations via rule *syncr* or *syncr′*. □

6.7.3 Properties α and β

We now proceed to analyze how equivalence $C \equiv D$ interacts with arbitrary reduction steps $(C \Longrightarrow C')$. We follow Huet's [10] strategy of decomposing global confluence into two properties, α and β. Informally, property α states that local confluence holds for two independent reduction steps starting from a single configuration, and property β states that a single reduction step on a configuration C can be emulated in an equivalent configuration D, preserving equivalence between C and D. The full generality of Huet's α and β are not needed; we present stronger analogues of α and β which are also satisfied by the operational semantics.

Lemma 6.18 ((β): Reduction on $\equiv$ Configurations) *If $C \equiv P$ and $C \Longrightarrow D$, then there exists Q such that $P \Longrightarrow^* Q$.*

$$C \equiv P \quad \wedge \quad C \Longrightarrow D \quad \Longrightarrow \quad \exists Q\ .\ \ P \Longrightarrow^* Q \quad \wedge \quad D \equiv Q$$

Proof: Without loss of generality, we may assume C has the form $C, \langle r : M\rangle$ (or $C, \langle l : E\rangle$) and that the reduction $C \Longrightarrow D$ acts on process r (or l). The proof is by cases on the judgment $C \Longrightarrow D$. Some representative cases are shown:

Case:

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

$\langle l : F\rangle \in P$ and $\mathcal{S}[\,V_1\ V_2\,] \equiv F$	Assumption, Definition
$\mathcal{S}[\,V_1\ V_2\,] \equiv \mathcal{S}'[\,V_1'\ V_2'\,]$	
and $V_1\ V_2 \equiv V_1'\ V_2'$	Equiv. and Ev. Context Lemma
$V_1 \equiv V_1'$ and $V_2 \equiv V_2'$	Inversion (cong. rule)
$V_1' = \lambda \mathtt{x} : A\,.\,M''$ or $V_1' = r$	Inversion Lemma

Subcase: $V_1' = \lambda \mathtt{x} : A\,.\,M''$

$\langle l : \mathcal{S}'[\,V_1'\ V_2'\,]\rangle \Longrightarrow \langle l : \mathcal{S}'[\,[V_2'/\mathtt{x}]M''\,]\rangle$	Reduction (rule *app*)
$M' \equiv M''$	Inversion (cong. rule)
$[V_2/\mathtt{x}]M' \equiv [V_2'/\mathtt{x}]M''$	Substitution Prop.
$\mathcal{S}[\,[V_2/\mathtt{x}]M'\,] \equiv \mathcal{S}'[\,[V_2'/\mathtt{x}]M''\,]$	Equivalence (cong. rule(s))
Hence $D \equiv Q$	

Subcase: $V_1' = r$

$\langle r : V\rangle \in P$ and $\lambda \mathtt{x} : A\,.\,M' \equiv V$	Inversion (rule *trans*$'$)
$\langle r : V\rangle \Longrightarrow^* \langle r : \lambda \mathtt{x} : A\,.\,M''\rangle$ and $\lambda \mathtt{x} : A\,.\,M' \equiv_o \lambda \mathtt{x} : A\,.\,M''$	Convertibility Lemma
$\lambda \mathtt{x} : A\,.\,M' \equiv \lambda \mathtt{x} : A\,.\,M''$	Definition ($\equiv_o$)
$\langle r : \lambda \mathtt{x} : A\,.\,M''\rangle, \langle l : \mathcal{S}'[\,V_1'\ V_2'\,]\rangle$ $\Longrightarrow \langle r : \lambda \mathtt{x} : A\,.\,M''\rangle, \langle l : \mathcal{S}'[\,\lambda \mathtt{x} : A\,.\,M''\ V_2'\,]\rangle$	Reduction (rule *syncr*$'$)

Then proceed as in case $V_1' = \lambda \mathtt{x} : A\,.\,M''$.

Case:

$$\frac{V\ \mathtt{tvalue}}{\langle r' : V\rangle, \langle l : \mathcal{S}[\,r'\,]\rangle \setminus \psi \Longrightarrow \langle r' : V\rangle, \langle l : \mathcal{S}[\,V\,]\rangle \setminus \psi}\ \mathit{syncr}'$$

$C \equiv P$	Assumption
$C \equiv D$	Synch. Pres. Equivalence
$D \equiv P$	Symmetry, Transitivity

Case:

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

$\langle l : E\rangle \in P$ and $\mathcal{S}[\,\texttt{let box}\,\texttt{u}\,\texttt{=}\,V\,\texttt{in}\,N\,] \equiv E$	Assumption, Definition
$E = \mathcal{S}'[\,\texttt{let box}\,\texttt{u}\,\texttt{=}\,V'\,\texttt{in}\,N'\,]$	Equiv. and Ev. Context Lemma
$V \equiv V'$ and $N \equiv N'$	Inversion (cong. rule)
$V = \texttt{box}\,M$	Assumption
$V' = \texttt{box}\,M'$ or $V' = r$	Inversion Lemma

Subcase: $V' = \texttt{box}\,M'$

$\langle l : \mathcal{S}'[\,\texttt{let box}\,\texttt{u}\,\texttt{=}\,\texttt{box}\,M'\,\texttt{in}\,N'\,]\rangle \Longrightarrow \langle r' : M'\rangle, \langle l : \mathcal{S}'[\,[\![r'/\texttt{u}]\!]N'\,]\rangle$	
	Reduction (rule *letbox*′)
$M \equiv M'$	Inversion (cong. rule)
$r' \equiv r'$	Equivalence (rule *eqloc*)
$N \equiv N'$	Assumption, Reflexivity
$[\![r'/\texttt{u}]\!]N \equiv [\![r'/\texttt{u}]\!]N'$	Substitution Prop.
$\mathcal{S}[\,[\![r'/\texttt{u}]\!]N\,] \equiv \mathcal{S}'[\,[\![r'/\texttt{u}]\!]N'\,]$	Equivalence (cong. rule(s))
Hence $D \equiv Q$	

Subcase: $V' = r$

$\langle r : V''\rangle \in P$ and $\texttt{box}\,M \equiv V''$	Inversion (rule *trans*′)
$\langle r : V''\rangle \Longrightarrow^* \langle r : \texttt{box}\,M'\rangle$ and $\texttt{box}\,M \equiv_o \texttt{box}\,M'$	Convertibility Lemma
$\texttt{box}\,M \equiv \texttt{box}\,M'$	Definition of $\equiv_o$
$\langle r : \texttt{box}\,M'\rangle, \langle l : \mathcal{S}'[\,\texttt{let box}\,\texttt{u}\,\texttt{=}\,r\,\texttt{in}\,N'\,]\rangle$	
$\quad\Longrightarrow \langle r : \texttt{box}\,M'\rangle, \langle l : \mathcal{S}[\,\texttt{let box}\,\texttt{u}\,\texttt{=}\,\texttt{box}\,M'\,\texttt{in}\,N'\,]\rangle$	
	Reduction (rule *syncr*′)

Then proceed as in case $V' = \texttt{box}\,M'$.

Case:

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

$\langle l : E\rangle \in P$ and $\texttt{let dia}\,\texttt{x}\,\texttt{=}\,V\,\texttt{in}\,F \equiv E$	Assumption, Definition
$\langle l' : V^{*\prime}\rangle \in P$ and $V^* \equiv V^{*\prime}$	Assumption, Definition

$E = \texttt{let dia}\,\texttt{x}\,\texttt{=}\,V'\,\texttt{in}\,F'$ Inversion
$V \equiv V'$ and $F \equiv F'$ Inversion (cong. rule)
$V = \texttt{dia}\,l'$ Assumption
$V' = \texttt{dia}\,l'$ or $V' = r$ Inversion Lemma

Subcase: $V' = \texttt{dia}\,l'$

$\langle l : \texttt{let dia}\,\texttt{x}\,\texttt{=}\,V'\,\texttt{in}\,F'\rangle, \langle l' : V^{*'}\rangle$
$\quad \Longrightarrow \langle l : l''\rangle, \langle l' : V^{*'}\rangle, \langle l'' : \langle\!\langle V^{*'}/\texttt{x}\rangle\!\rangle F'\rangle$
Reduction (rule *syncl*)
$l'' \equiv l''$ Equivalence (rule *eqloc*)
$V^* \equiv V^{*'}$ Assumption, Reflexivity
$\langle\!\langle V^*/\texttt{x}\rangle\!\rangle F \equiv \langle\!\langle V^{*'}/\texttt{x}\rangle\!\rangle F'$ Substitution Prop.
Hence $D \equiv Q$

Subcase: $V' = r$

$\langle r : V''\rangle \in P$ and $\texttt{dia}\,l' \equiv V''$ Inversion (rule *trans'*)
$\langle r : V''\rangle \Longrightarrow^* \langle r : \texttt{dia}\,l'\rangle$ and $\texttt{dia}\,l' \equiv_o \texttt{dia}\,l'$ Convertibility Lemma
$\texttt{dia}\,l' \equiv \texttt{dia}\,l'$ Definition of $\equiv_o$
$\langle r : \texttt{dia}\,l'\rangle, \langle l : \texttt{let dia}\,\texttt{x}\,\texttt{=}\,r\,\texttt{in}\,F'\rangle$
$\quad \Longrightarrow \langle r : \texttt{dia}\,l'\rangle, \langle l : \texttt{let dia}\,\texttt{x}\,\texttt{=}\,\texttt{dia}\,l'\,\texttt{in}\,F'\rangle$
Reduction (rule *syncr'*)
Then proceed as in case $V' = \texttt{dia}\,l'$.

□

Lemma 6.19 ((α): Local Confluence (modulo $\equiv$)) *If $C \Longrightarrow C_1$ and $C \Longrightarrow C_2$, then there exist D and D' (where $D \equiv D'$) such that $C_1 \Longrightarrow^* D$ and $C_2 \Longrightarrow^* D'$.*

$$C \Longrightarrow C_1 \;\wedge\; C \Longrightarrow C_2 \quad\Longrightarrow\quad \exists D, D' \;.\; \; D \equiv D' \;\wedge\; C_1 \Longrightarrow^* D \;\wedge\; C_2 \Longrightarrow^* D'$$

Proof: We will consider pairs of such transitions $C \overset{\alpha(w)}{\Longrightarrow} C_1$ and $C \overset{\beta(\sigma)}{\Longrightarrow} C_2$, where $\alpha(w)$ denotes application of rule α to process w. The reduction rules fall naturally into certain classes (silent, local, and spawn) with properties as stated in the table below. The forms of C and C' are given for reduction of a process $\langle r : M\rangle$ though of course reduction of a process $\langle l : E\rangle$ is also possible.

Class	Rule	Form of C and C'
Silent	$\mathit{syncr}, \mathit{syncr}'$	$C \equiv C'$
Local	$\mathit{app}, \mathit{app}', \mathit{letdia}$	$C = C_1, \langle r : M \rangle \ \wedge\ C' = C_1, \langle r : M' \rangle$
Spawn	$\mathit{letbox}, \mathit{letbox}', \mathit{letbox}_p, \mathit{syncl}$	$C = C_1, \langle r : M \rangle \ \wedge\ C' = C_1, \langle r : M' \rangle, C_2$

Not all combinations of two transitions $C \overset{\alpha(w)}{\Longrightarrow} C_1$ and $C \overset{\beta(w')}{\Longrightarrow} C_2$ are possible. Because evaluation contexts are uniquely determined (excluding synchronization contexts such as $\mathcal{R}[\,r'\,]$), in many cases $\alpha(w)$ and $\beta(w')$ occur in separate processes ($w \neq w'$). We argue that reductions in separate processes do not interfere and that equivalence can be re-established by performing reductions $\beta(w')$ and $\alpha(w)$ on C_1 and C_2, respectively. We consider a few representative combinations the three classes of reduction steps:

(Silent, Silent) In this case, $C \overset{\alpha(w)}{\Longrightarrow} C_1$ and $C \overset{\beta(w')}{\Longrightarrow} C_2$. Now $C \equiv C_1$ and $C \equiv C_2$ by the lemma stating that synchronization preserves equivalence. We conclude $C_1 \equiv C_2$ by symmetry and transitivity, with no further reduction steps required. The result holds even if $w = w'$, that is, if α and β apply to the same process w.

(Silent, Local) Without loss of generality, assume $C \overset{\alpha w}{\Longrightarrow} C_1$ is the silent step. Then $C \equiv C_1$. By the lemma regarding reduction on equivalent configurations, we can replicate the effect of $C \overset{\beta(w')}{\Longrightarrow} C_2$, with some sequence of reductions $C_1 \Longrightarrow^* D$ such that $D \equiv C_2$. The same result holds if $w = w'$.

(Silent, Spawn) Similar to (Silent,Local).

(Local, Local) We assume $C \overset{\alpha(w)}{\Longrightarrow} C_1$ and $C \overset{\beta(w')}{\Longrightarrow} C_2$. By the lemma stating that evaluation contexts are "uniquely" determined (excluding redices r), a combination of two local reductions is only possible if they occur in separate processes ($w \neq w'$). Hence it will be possible to perform $\alpha(w)$ to make a step $C_2 \overset{\alpha(w)}{\Longrightarrow} D$ and $\beta(w')$ to make a step $C_1 \overset{\beta(w')}{\Longrightarrow} D$. This is because C_1 and C_2 remain syntactically identical to C except for the particular processes (w, w') affected by the initial steps $C \Longrightarrow C_1$ and $C \Longrightarrow C_2$. Since the second step will be made in a different process, it remains applicable. Both sequences $\alpha\beta$ and $\beta\alpha$ yield the same result D.

(Local, Spawn) As before, $\alpha(w)$ and $\beta(w')$ must occur in separate processes. Performing $C \overset{\beta(w')}{\Longrightarrow} C_2$ spawns a new process with a fresh label. This new process will not interfere with reduction step $\alpha(w)$ because it has a fresh label. Hence $C_2 \overset{\alpha(w)}{\Longrightarrow} D$. It will also be possible to make the transition $C_1 \overset{\beta(w')}{\Longrightarrow} D$, choosing the same fresh label for the newly spawned process. As before, α and β commute, yielding the same result configuration D.

(Spawn, Spawn) This case is similar to (Local,Spawn) except that two new processes are created. Assume $C \overset{\alpha(w)}{\Longrightarrow} C_1$ and $C \overset{\beta(w')}{\Longrightarrow} C_2$. In the case of *letbox* (and variants), these new processes do not interfere with $C_1 \overset{\beta(w')}{\Longrightarrow} D$ nor with $C_2 \overset{\alpha(w)}{\Longrightarrow} D$. In the case of two *syncl* reductions, the fact that we duplicate the process $\langle l' : V^* \rangle$ is essential to ensure that α and β do not interfere.

6.7.4 Global Confluence

Having established property α (Local Confluence) and β (Reduction on Equivalent Configurations), we claim that the conjunction of these two are sufficient for global confluence (modulo $\equiv$). Therefore, the operational semantics satisfies:

Theorem 6.4 (Global Confluence (modulo $\equiv$)) *Assume ψ `csound` and both C and P are well-formed ($\psi \vdash^c C : \Lambda$ and $\psi \vdash^c P : \Lambda$). Then the following confluence property holds:*

$$C \equiv P \;\wedge\; C \Longrightarrow^* C' \;\wedge\; P \Longrightarrow^* P'$$

$$\Longrightarrow \quad \exists D, Q \;.\;\; D \equiv Q \;\wedge\; C' \Longrightarrow^* D \;\wedge\; P' \Longrightarrow^* Q$$

Proof: see [10]. Note that $\Longrightarrow$ satisfies the condition that all reduction sequences terminate. The strong normalization theorem applies because ψ is assumed to be sound. □

7 Why Modal Types?

Since the laws of modal logic are *designed* to characterize structures in which truth of propositions is localized, it is quite natural that constructive modal logic be based on proof objects with varying locality and mobility. The proofs of A `valid` are freely mobile terms, proofs of A `true` are locally available terms, and proofs of A `poss` represent remote, immobile terms. We hope to convince the reader that modal logic proof terms are a sort of universal calculus for distributed computation in the sense that the typing principles and much of the operational behavior of other distributed programming languages can be projected into modal logic and understood in terms of general logical principles.

Safe, statically typed, languages for distributed computation usually adopt at least some of the typing principles of modal logic. For example, the definition of valid proof terms ($\Delta; \cdot \vdash M : A$) in modal logic captures the idea that valid (mobile) terms may depend only on other valid (mobile) terms. Adoption of this principle seems inevitable, since operationally speaking, when moving an arbitrary term, bindings for its free variables must also be moved. Some languages place additional restrictions on the form of terms to be made mobile, allowing only values of function type (closures) to be marshaled, or in the extreme case, that only certain types of parameter value can be marshaled (requiring that the code be predistributed).

The principle that a valid (mobile) term is also available here ($\Delta; \Gamma \vdash \mathtt{u} : A$) reflects the idea that we can receive the result of a remote computation or interact with a proxy as if the remote entity were local. Though the calculus of modal logic distinguishes `u` from other terms, one can also hide this distinction. Some languages do not adopt the operational semantics of synchronization, instead, the remote term is represented by a local proxy. If the proxy implementation is powerful enough, behaving exactly as a local term would, this strategy is logically equivalent to synchronization.

Finally, the possibility fragment modal logic reflects the idea that some entities are immobile and possibly remote. The typing principle $\Diamond E$ describes how we may use such resources by sending mobile code to a particular location. Since we did not assume symmetry in accessibility, we cannot receive the result of such a computation. Furthermore, we may only use resources from a single location at a time, since these entities cannot be combined ($\Diamond A \times \Diamond B \not\Rightarrow \Diamond(A \times B)$). These principles resemble the concept of "one way" method calls sent to a remote object, or the behavior of a mobile process which chooses to move to a location with some known resources. Explicit recognition of these principles (separate from necessity) is more rare, since moving to a particular location is a special case of general mobility. Also, it is often possible to hide the fact that a resource is immobile and remote by implementing a local proxy.

Recognizing such principles in other distributed languages is often complicated by the fact that the spatial modalities are hidden, and conversions between local and mobile terms are made implicitly. Often, rather than providing a general mechanism such as $\Box I$ (the definition of necessity) to make terms mobile, only certain forms of code (for example closures of type $A \rightarrow B$) can be made mobile as long as such code depends only on values of "marshalable" type. The assertion "A is a marshalable type" then corresponds to selective adoption of a non-logical axiom $A \rightarrow \Box A$ (only for type A). Values of types B, for which such a marshaling axiom does not exist, are then effectively immobile.

It is often tempting to try to hide the logical distinctions between remote mobile, local, and remote immobile entities when designing and implementing a distributed language. However, there are some negative consequences of such an abstraction. Operationally, blurring these boundaries requires a heroic effort to make remote things appear to be local (and the converse). Simply marshaling everything by copying can lead to semantic anomalies, and overuse of proxies leads to inefficiency and unpredictable performance. Perhaps the best balance of abstraction and precision could be achieved when the calculus of modal logic is treated as an intermediate language. Another possibility is to use typing principles from modal logic in a locality analysis framework to recover the distinction between remote and local entities by type inference (see [12] for an example). These sorts of approaches could lead to a better understanding of distributed programs or more efficient implementations even if such distinctions are never revealed to the programmer.

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

We must keep in mind that there are two reasons to program with modal types $\Box A$ and $\Diamond A$ — safety and concurrency. From a logical point of view, $\texttt{box}\, M$, $\texttt{dia}\, E$ and their elimination forms provide a safe way to work with a mix of mobile and immobile entities. Though the generalized language does not have any primitive localized terms, we can see that locality of term values is respected by observing the typing principles for mobile code ($\Box I$ and $\Diamond E$). Since these constructs require mobile code to be closed with respect to Γ, we will never be forced to marshal arbitrary term values at runtime. From a behavioral point of view, the use of $\Box$ has a secondary effect of introducing concurrency. Mobility is somewhat intertwined with concurrency because we assume each abstract "location" has an independent capability to perform computation.

The calculus of proof terms supports two distinct programming styles.

The necessity fragment allows one to build boxed terms, spawn these terms for evaluation at an arbitrary location, and receive the result of such a remote computation as a local value. On the other hand, the possibility fragment allows one to compute locally with an expression of the form $\{M\}$ or jump to some other location (denoted by l) where a remote resource is available. The two programming styles are not incompatible because $\texttt{let box}\,\texttt{u} = M\ \texttt{in}\ F$ allows one to embed spawning of terms in the context of a jumping computation. However, programs which perform any such jumps will be expressions $E \div A$ due to the typing rules governing possibility.

8.1 Runtime Environments

In some cases one may want to program under the assumption that some library code, localized resources, or other information about the environment will be provided at runtime. In these cases, open programs can be typed under some initial assumptions $\Delta_0; \Gamma_0$. If we assume such an open program is placed as the process $\langle w_0 : P\rangle$, then the realizations of hypotheses in $\Delta_0; \Gamma_0$ should abide by the following restrictions:

Hypothesis	Typing	Form of Constraints
$\texttt{u} :: A$	$\Lambda_0 \backslash \psi_0 \vdash_{r\triangleleft} M : A$	$\psi_0 \vdash^a r_i \triangleleft r \ \wedge\ \psi_0 \vdash^a r \triangleleft w_0$
$\texttt{x} : A$	$\Lambda_0 \backslash \psi_0 \vdash_{w_0} M : A$	$\psi_0 \vdash^a r_i \triangleleft w_0 \ \wedge\ \psi_0 \vdash^a w_0 \triangleleft l_i$

We require that realizations of valid hypotheses $\texttt{u} :: A$ be typed under the quantified typing judgment $\Lambda_0 \backslash \psi_0 \vdash_{r\triangleleft} M : A$, whereas the locally true hypotheses $\texttt{x} : A$ are only required to be well-formed at the particular location w_0. Realizing $\texttt{u} :: A$ at r may impose some constraints on the location of r relative to some number of r_i on which it depends. Realizing $\texttt{x} : A$ at w_0 imposes constraints on the location w_0 relative to some number of r_i and l_i on which it depends. Consequently, we see that programs $\langle w_0 : P\rangle$ must be placed (conceptually) in a certain relation to the resources on which they depend. We also note that location labels l, corresponding to hypotheses of logical possibility, are not allowed to occur in realizations of $\texttt{u} :: A$. It is, however, possible to provide a location label in a realization of $\texttt{x} : A$.

We can then place closed terms corresponding to $\texttt{u}$, as independent processes of the form $\langle r_i : M_i\rangle$, substituting labels r_i for $\texttt{u}$ and terms for $\texttt{x}$ into the program. It is also possible to substitute realizations of $\texttt{u} :: A$ directly if desired. This leads to an initial configuration $\langle r_1 : M_1\rangle, \langle r_2 : M_2\rangle, \ldots, \langle w_0 : P\rangle \setminus \psi_0$.

To complete the picture, we should also consider processes of the form $\langle l_i : V_i^*\rangle$ and their meaning. Such processes are a way to represent remote localized resources present in the runtime environment. They are useful in conjunction with hypotheses $\mathtt{x} : \Diamond A$ realized by $(\mathtt{dia}\, l_i)$. By using $\mathtt{x} : \Diamond A$, the program can then jump to the location l_i and resume computation in a setting where a term of type A is locally available.

Generally, a configuration will consist of processes of both kinds. Initially, processes $\langle r_i : M_i \rangle$ can be viewed as globally available, mobile resources, and processes $\langle l_j : E_J \rangle$ as localized resources, fixed to a particular location. The program is introduced as a process $\langle w_0 : P \rangle$.

$$\langle r_1 : M_1 \rangle, \ldots, \langle r_i : M_i \rangle, \langle w_0 : P \rangle, \langle l_1 : E_j \rangle, \ldots, \langle l_j : E_j \rangle \setminus \psi_0$$

As the process configuration evolves, additional processes $\langle r : M \rangle$ can be spawned. These $\langle r : M \rangle$ are mobile and can be placed at distinct locations, though the scope of r is not global as before. Duplicates of processes $\langle l : E \rangle$ are created as the program P jumps between locations l, though all duplicates of a particular $\langle l : E \rangle$ should be regarded as sharing the same fixed location.

To take full advantage of localized resources present in the runtime environment, it will be necessary to encode the resources at each location as an assumption $\mathtt{x} : \Diamond(A_1 \times A_2 \times \ldots \times A_k)$. Further jumps to other locations will only be allowed if one or more resources in $\Diamond(A_1 \times A_2 \times \ldots)$ permit it, since the typing rule for $\Diamond E$ requires we drop all other locally true assumptions Γ. For example $\Diamond(A_1 \times A_2 \times \Diamond(B_1 \times B_2))$ would allow a program to go to the location of $(A_1 \times A_2 \times \Diamond(B_1 \times B_2))$ perform some computation with A_1 and A_2, then jump to the location of $(B_1 \times B_2)$ and continue. In the general case, a directed acyclic graph of locations and resources can be encoded with possibility and products.[9]

8.2 Definition of Recursion

Many interesting distributed programs require recursion to specify. These programs can be characterized as having a variable degree of parallelism. That is, they may "unroll" at runtime to a tree-structured or looping form of computation involving a variable, possibly unbounded number of worlds. Corresponding to the distinction between valid, mobile hypotheses $(\mathtt{u} :: A)$ and local hypotheses $(\mathtt{x} : A)$, we introduce two forms of fixpoint over terms. $\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M$ is referred to as valid or mobile fixpoint and $\mathtt{fix}\,(\mathtt{x} : A)\,.\,M$

[9]This limitation is due to the requirement that accessibility constraints be acyclic. We are considering how best to represent sets of interaccessible locations, so that more flexible jumping behavior can be supported.

as local fixpoint.

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

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

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

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

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

$$\frac{}{\langle r : \mathcal{R}[\,\mathtt{fix}\,(\mathtt{x} : A)\,.\,M\,]\rangle \setminus \psi \Longrightarrow \langle r : \mathcal{R}[\,[\mathtt{fix}\,(\mathtt{x} : A)\,.\,M/\mathtt{x}]M\,]\rangle \setminus \psi}\ \mathit{fix}$$

$$\frac{}{\langle r : \mathcal{R}[\,\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M\,]\rangle \setminus \psi \Longrightarrow \langle r : \mathcal{R}[\,[\![\mathtt{fix}_v\,(\mathtt{u} :: A)\,.\,M/\mathtt{u}]\!]M\,]\rangle \setminus \psi}\ \mathit{fix}_v$$

There are, of course, variants fix' and fix'_v for reducing fixpoint terms in an expression evaluation context (such as $\mathcal{S}[\,\mathtt{fix}\,(\mathtt{x} : A)\,.\,M\,]$). Type preservation and progress proofs for the operational semantics can be extended to account for fixpoint. In the case of the progress theorem, we note that fixpoint is not a value, but that we can always apply one of the rules fix or fix_v. In the case of the type preservation theorem, a substitution property will ensure proper typing of the result.

One may also consider recursion over expressions. For reasons of conceptual economy and uniformity, we adopt an approach of encoding such fixpoints with fix_v or fix. For example, $\mathtt{fix}_v\,(\mathtt{loop} :: \Diamond A)\,.\,\mathtt{dia}\,E$ binds a fixpoint variable $(\mathtt{loop} :: \Diamond A)$ in E. In the body E, the one can use the idiom `let dia result = loop in` F_l to perform a recursive jump to an unrolled copy of E. When and if E terminates without making such a nested jump, we continue with F_l. Note that the fixpoint body must be Γ-closed, a consequence of using (`loop` $:: \Diamond A$) rather than a local fixpoint variable (`loop` $: \Diamond A$).

```
let c =
    fixv loop .
        dia
            (* fixpoint body *)
            ...
            (* recursive call *)
            let dia result = loop in
                (* continuation after return *)
                $F_l$
in

let dia result = c in $F_c$
```

The fixpoint body occurring before the recursive call might consist of a sequence of local computations (`let dia x = dia` $\{M\}$ `in` ...) or jumps to other locations (`let dia x = dia` l' `in` ...) It seems clear that valid or mobile fixpoint over (`u` $:: \Diamond A$) is a useful idiom. But local fixpoints `fix` (`x` $: \Diamond A$) `. dia` E do not seem to be useful, since the scope of (`x` $: \Diamond A$) is so limited.

8.3 Globally Accessible Locations

Fixpoint over expressions should allow us to jump repeatedly between distinct locations represented by a set of globally mobile hypotheses ($\mathtt{v_i} :: \Diamond A_i$) in Δ. But how can such mobile assumptions of type $\Diamond A_i$ be realized? Essentially, the difficulty is that we can conclude $\Lambda\backslash\psi \vdash_w \mathtt{dia}\, l : \Diamond A$, but *not* $\Lambda\backslash\psi \vdash_{w\lhd} \mathtt{dia}\, l : \Diamond A$. To make the latter conclusion sound, we must know that l is accessible from *any* other location.

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

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

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

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

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

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

The rule permits typing of l in the context of a mobile term or expression, where such occurrences were not typeable before. For example, we may now conclude $\Lambda\backslash\psi; \Delta; \cdot \vdash_{w\lhd} \mathtt{dia}\, l : \Diamond A$, which allows us to realize assumptions $\mathtt{u} :: \Diamond A$ in Δ_0.

Recursion among processes is now permitted under configuration typing. For example $\langle l : l'\rangle, \langle l' : l\rangle$ is permitted under $\psi = \forall\mathtt{w}\,.\,\mathtt{w} \lhd l \wedge \forall\mathtt{w}\,.\,\mathtt{w} \lhd l'$. But it is intended that this feature be used judiciously to represent the initial distributed environment in which a program runs. As mentioned above, such locations l could hold bindings for global resources ($\mathtt{v_i} :: \Diamond A_i$). Programmers cannot create such cyclic structures, since none of the rules of the operational semantics introduce this form of constraint.

8.4 Axioms of Modal Logic (S4)

Below are reproduced the axioms of S4, together with their realizations as proof terms. It is interesting to consider what behaviors such proofs correspond to in the setting of distributed computation.

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

Axiom RB captures the behavior of spawning a boxed term for evaluation, and receiving the value of that computation for local use. Axiom DD shows us how to apply a boxed (mobile) function to a localized term of type $\Diamond A$. The function $\Box(A \to B)$ is made mobile with letbox, then it can be received as `u` and applied $\{\texttt{u z}\}$ to the localized value `z` of type A.

We may compose axioms DB and RB to obtain $\Box(A \to B) \to \Box A \to B$. Axiom DB constructs a boxed term applying the function $\Box(A \to B)$ to a mobile argument $\Box A$. Axiom RB then spawns the function application, making the result B available. Note that the axioms relating to $\Diamond$ do not exhibit behavior immediately, because $\texttt{dia}\,E$ is a value encapsulating a localized computation. The value of such expressions is obtained by forcing evaluation with $\texttt{let dia x =}\,M\,\texttt{in}\,F$, causing a shift in our perspective. Understood in this way, axiom TD (or a generalization) shows how to encapsulate a series of such "jumps" between locations as a single one.

8.5 Example (Marshalling)

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

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

In cases such as this, when the boxed term is already a value, it would be highly desirable to inline the operation let box u = box V in (. . .). By this we mean simply performing the substitution $[\![V/\mathtt{u}]\!]$ without generating an intermediate process. This is consistent with the intuition that $\Box A$ captures mobility – the value V may move, but is not *forced* to move to some remote location.

8.6 Example (Concurrency)

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

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

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

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

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

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

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

8.7 Example (Localized Resources)

In the possibility fragment of the language there is no actual movement without primitive remote resources $\Diamond A$ represented by $\texttt{dia}\,l$. In this example, we use each $\Diamond A$ to describing some location or environment providing I/O primitives encapsulated as functions $A \to \bigcirc B$. Here $\bigcirc B$ is the monadic type of computations producing B. One could give a detailed account of the integration of effects with $\Box$ and $\Diamond$, but we do not do so here. In this example, the particular I/O operations relate to submitting print jobs, querying for the status of a job, reading local files, etc. Assume the following variables in the global context Δ:

```
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. The $\bigcirc$ elimination construct `let comp` x = M in E denotes sequential composition of M with E. That is, the computation denoted by M is evaluated to produce a value V which is bound to `x` in E.

```
let dia h_env = home in

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

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

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

9 Related Work

Fundamentally, our work is an attempt to uncover simple, logical principles underlying distributed computation. We believe the critical questions are these: What are the local resources that distinguish locations from one another? And where may fragments of code (which might depend on these resources) be executed safely? One can address these questions either in the setting of a new primitive calculus for distributed computation, or by considering what runtime support structures are necessary to implement mobility in a more conventional programming language. We have chosen the former, believing that the foundational approach will yield clear principles and more generally applicable results. But it is also important to consider when and how logical principles show up in applied settings.

Jia and Walker [11] have taken such a foundational approach in their work on λ_{rpc} , a calculus for distributed computation. The methodology and motivation behind λ_{rpc} is quite similar to this work; the authors adopt a spatial interpretation of modal logic, reading propositions as types and proof terms as programs. However, the logical basis of their calculus extends beyond a minimal pure modal logic, incorporating certain hybrid logic features that allow worlds and the accessibility relation to be referenced explicitly as places and edge names. Jia and Walker reach similar conclusions about the meaning of $\Box A$ and $\Diamond A$ types and their role in a distributed computation. But differences between their logical formalism and that of Pfenning and Davies [13] lead naturally to a qualitatively different programming model from the calculus of S4 we develop in this paper. These differences are discussed in detail below.

Other researchers adopting the foundational approach have based their work on a process calculus, such as the Pi or join calculus. These sorts of calculi model the connectivity of processes, but not location and localized resources in an explicit sense. A notion of location is then added to the operational semantics, the language is extended with one or more primitives for mobility, and (optionally) restrictions are imposed on how and where a process may safely move. Since process calculi usually allow changing the scope of channel names by name passing and scope extrusion, some means of restricting or monitoring the flow of names is crucial. Without such restrictions, all names are potentially mobile and one cannot enforce any stable notion of locality. Cardelli and Gordon [6, 7] restrict mobility with a specification-logic (with classical semantics) for ambient calculus terms. Hennessy, *et. al.* [9] take a type-based, constructive approach in which names are inherently associated with a location. We explain these approaches below.

Issues of mobility and locality also arise when one considers how to interpret a more conventional programming language in a distributed setting. We discuss a type-based locality analysis framework due to Moreira [12] for determining which values (and references in particular) "escape" to other locations. Though the essence of locality and mobility is somewhat obscured in this setting, some of the principles of modal logic seem to show up in restricted forms.

9.1 λ_{rpc} Calculus

As mentioned above, the most closely related work is the λ_{rpc} calculus developed independently and concurrently by Jia and Walker [11]. The type system of λ_{rpc} is also inspired by a spatial interpretation of modal logic, though it is an extension of S5 (not S4) with some hybrid-logic features. The hybrid-logic aspect of the λ_{rpc} type system, which introduces explicit worlds and edge names, makes comparison to a standard modal logic difficult. We also believe these features have consequences for the generality or portability of programs across different runtime environments.

The terms of λ_{rpc} are intrinsically typed and annotated throughout with types τ, places p, and edge names n. By drawing place names from set P, a program may refer to some finite number of sites for computation, with the roles of these sites being assigned by the programmer. Loosening this limitation somewhat, Jia and Walker introduce the edge names which give the programmer flexibility in naming locations $z.n$ defined relative to z. That is, $z.n$ denotes the place obtained by following edge n from z. So given a root location p_0, and edge names n_i drawn from a set N the programmer can refer to an infinite variety of locations $p_0.n_i$, $p_0.n_j$, $p_0.n_i.n_j$, etc. However, the structure of computations is still quite rigid in that all computation occurs at a definite location, be it specified absolutely (p) or relatively ($z.n$). For example, the $\Box$ elimination construct (`bc` e_1 `at` z `as` x `in` e_2) is labeled with z, specifying the location at which e_1, having type $\Box\tau$, is to be evaluated. Most other terms of the language are similarly annotated.

So the type system and operational semantics of λ_{rpc} make it a very expressive calculus, allowing the programmer to place terms and perform computations in particular locations. This aspect of the calculus allows one to implement algorithms specialized to a particular number of nodes or network topology. Thus it seems that λ_{rpc} and our calculus of S4 solve different problems. λ_{rpc} is a more precise, low-level language which, by design, reveals aspects of the distributed environment to the programmer. On the other hand, our calculus assumes very little about the runtime environment and hides such details from the programmer. In our calculus, definite locations are represented as $\mathtt{u_i} :: \Diamond A_i$, representing resources of type A_i at some abstract, hidden locations.

9.2 Mobile Ambients

The ambient calculus, as developed by Cardelli and Gordon [5], is a novel form of process calculus based on ambients (locations) rather than channels. The ambient notation $n[\dots]$ allows representation of location in process configurations. Simultaneously, ambients facilitate communication by providing a space in which processes may exchange messages (replacing the concept of channels).

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

In subsequent papers [7, 1, 2], an "ambient logic" is developed to characterize the behavior and spatial distribution of processes. Ambient logic is not intended to be a system for assigning types to processes. Rather, it is a language for making statements about a given process configuration (considered as a model for the logic). These propositions are then either satisfied by the given model, or not, according to the semantics of the logic. Ambient logic includes modal operators $\Box, \Diamond$ of both the spatial and temporal variety which are interpreted by reference to spatial (hierarchical inclusion) and temporal (reduction steps) notions of accessibility. The full ambient logic is very precise, and allows one to specify undecidable properties of a program. Verifying a formula in decidable fragments of ambient logic can be accomplished by model-checking.

Notably, Cardelli and Gordon [6] have extended ambient logic with propositions expressing hiding, revelation, and freshness of names in order to characterize the scope and mobility of names. However, since processes in the ambient calculus are not inherently required to preserve locality of names, most name-hiding properties must be formulated and proved (by model-checking) relative to a particular implementation.

9.3 DPI and Process Typing

Hennessy, *et. al.* have developed a variant of the Pi-calculus, called DPI, suitable for exploring issues of locality and mobility. It extends the Pi-calculus with a notation for process location, and a simple $\texttt{go}\ l\ .\ P$ action which moves P to location l where execution of P resumes.

The typing systems developed for this language are described in papers by Hennessy, Riely, Yoshida, and others [9, 15, 8]. Their types and judgments, though not modal ($\Box, \Diamond$), do make reference to "worlds", represented with an ambient-like notation $l[\dots]$.

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

Informally speaking, we can find counterparts to some modal types in the scheme of location types. For example, a term of type $\Box A$ corresponds to a process P which is well-formed in a location of type $\texttt{loc}\{\}$ (the top type of the location typing hierarchy). Such a process may move to any location, since it depends on no local names. General terms of type $\Diamond A$ do not have a direct analogue in the DPI typing system, since processes cannot be removed from the context in which they are well-formed, but for the special case of channel names, $\Diamond A$ corresponds to the use of existential types $A@L$ (there exists a location L in which A) to characterize channel names which "escape" their original scope of definition.

Interestingly, the behavior of our $\texttt{let dia}\,\texttt{x} = M\,\texttt{in}\,F$ construct defined in this paper is quite similar to $\texttt{go}\ l\ .\ P$, in the sense that F (and P) are being sent to a new location. The difference is that $\texttt{let dia}\,\texttt{x} = \texttt{dia}\,E\,\texttt{in}\,F$

allows F access only to the *value* of E, rather than all the resources in scope at E's location. This is a natural outcome, given that our language is oriented toward evaluation rather than interaction.

9.4 Locality Analysis of References

As was discussed in one of the examples, it is possible to use the system of modal types to localize references and prevent them from escaping the location where they were created. Other work by Moreira [12] has addressed this particular problem in detail, though not by taking the point of view that references are localized. Instead, Moreira develops a type-based system for locality analysis which can (in some cases) distinguish between references used only locally, and those which escape to processes running on other machines. Both forms of reference are considered permissible and are type-safe, though access to a purely local reference can be optimized. In cases when the analysis cannot infer with certainty that a reference is local, it is conservatively assumed to be mobile.

Though we took the point of view that references are characterized by logical possibility, some aspects of Moreira's treatment of references can be understood in the necessity fragment of modal logic. Assume, for a moment, that one considers all reference cell primitives to be terms (and hence potentially mobile or escaping). A reference cell could then be boxed ($\texttt{box}\,\texttt{ref}\,M$) and made available as a valid hypothesis $\texttt{u} :: A\,\texttt{ref}$. Such a $\texttt{u}$ would be a sort of explicitly escaping reference.

Since we were not particularly interested in tracking which terms were mobile, the typing principle $(\Delta, \texttt{u} :: A, \Delta'; \Gamma \vdash \texttt{u} : A)$ derived from S4's assumption of reflexivity was used. Shifting to a locality analysis requires changing the properties of the logic to maintain the distinction between validity and truth, disallowing $\Box A \to A$ (at least for certain types A). Moreira's notion of labeled types then corresponds to the distinction between ordinary typing $\vdash M : A$ and a new explicit validity judgment $\vdash M :: A$. Since reflexivity is thus eliminated, synchronization is no longer a logically acceptable operational interpretation of the valid hypothesis $\texttt{u}$. The reference cell primitives must now interact with $r :: A\,\texttt{ref}$ as a local proxy for an escaped, mobile reference. The typing rules for primitive operations are tricky, since updating a reference cell can become a back-channel way of making terms mobile without box/letbox. It is possible to protect them, as Moreira did, by distinguishing locality ($M :: A$ versus $M : A$) when typing the primitive operations. For example, we should not allow $M \,\texttt{:=}\, N$ when $N : A$ is local but the reference $M :: A\,\texttt{ref}$ is mobile.

Though shifting to mobile references and locality analysis required adjustments to the logic and calculus, some of Moreira's typing principles appear to have more direct analogues in modal logic. For example, the **esc**? predicate for placing locality constraints on the free variables of function terms seems to be built into the typing rule for $\mathtt{box}\, M$ as the idea that proofs of A `valid` (mobile terms) can only depend on valid hypotheses (other mobile terms).

Much of the complexity of locality analysis seems to arise from the requirement that the distinction between local and escaping terms be transparent to the programmer. The language of modal logic is a sort of primitive calculus which makes such properties explicit. Entities with differing locality and mobility have distinct syntactic forms and types, which is a conceptual advantage, if not a practical one.

10 Conclusion and Future Work

Starting from an intuitionistic formulation of modal logic, we considered the proof terms for that logic as a programming language. The classical notions of worlds and accessibility were reflected concretely as processes $\langle r : M\rangle$ and $\langle l : E\rangle$ and accessibility constraints ψ, with accessibility governing the dependencies between processes. At the term level, we found the natural and type-sound operational interpretation for type $\Box A$ to be a boxed (mobile) term, with $\Box$ elimination spawning such a mobile term for evaluation at an arbitrary new location. The relationship between validity and truth, characterized by $A\ \mathtt{valid} \vdash A\ \mathtt{true}$, corresponded to the ability to receive the result of such a computation at all other accessible locations. The interpretation of $\Diamond A$ was as a local representation of a remote, immobile term. When $\Diamond A$ is derived through reflexivity ($A\ \mathtt{true} \vdash A\ \mathtt{poss}$), this is merely a way of hiding locality, forcing a term to become immobile. In cases when the encapsulated resource is truly remote, elimination of $\Diamond A$ was interpreted as a "jump" to the location of that resource for further computation.

We have shown that the modal types $\Box A$ and $\Diamond A$ are a safe and natural way to mix mobility and localized resources in a distributed computation. The necessity and possibility fragments of the language interact to enforce some restrictions on how and where certain terms are available. Mobility is permitted only for terms closed with respect to locally true hypotheses (or almost so, in the case of $\mathtt{let\ dia}\, \mathtt{x} = M\, \mathtt{in}\, F$). These restrictions on the scope of locally true hypotheses ($\mathtt{x} : A$) and the fact that we are not allowed to pass arbitrarily from possibility to truth are the essential characteristics of modal logic which ensure that local values never "escape" and become mobile.

The core modal calculus of proof terms is a sort of schema for programming with mobile and immobile things, but is underdeveloped with respect to concrete examples of localized resources. While $\{M\}$ and processes of the form $\langle l : \{M\}\rangle$ can simulate a localized resource; this immobility is self-imposed and not *intrinsic* to the term M. For any closed term M of the pure language, we might conclude $\texttt{box}\, M : \Box A$ as well. It should be valuable to explore extensions of the calculus with types types and terms representing truly *localized* entities. We believe effectful computations are a natural example to use, since the semantics of effects depend on a local machine state which is often quite difficult to move or replicate at runtime.

Also in future work, we plan to consider role of world-structure and accessibility in more detail. Though the choice of S4 as a logical foundation leads to greater generality than S5, in the sense that no assumption of symmetric accessibility is present, it remains to be seen whether this neutrality has practical value. There is also a balance to be struck when deciding how much of the underlying world-structure to reveal to programmers through the language and its type system. If too much is revealed, programs can become rigid and specialized to a particular topology of locations determined by accessibility. If too little is revealed, programmers may be frustrated by the inability to force collocation of processes or otherwise control the layout of a distributed program. The relative utility of one system over another is difficult to establish definitively, but might be argued through experimentation and careful consideration of examples.

Acknowledgments: The author thanks Frank Pfenning for many discussions and guidance on this topic as well as Tom Murphy, Robert Harper, and Karl Crary for their feedback.

References

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

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

[3] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languagese and Programming, 26th International Colloquium (ICALP)*, volume 1644 of *LNCS*, pages 230–239. Springer, 1999.

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

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

[6] 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.

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

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

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

[10] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *JACM*, 27(4):797–821, October 1980.

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

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

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

[14] Davide Sangiorgi. Termination of processes. Applies method of logical relations to prove termination for a fragment of the Pi calculus, December 2001.

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