A multi-modal type system and its procedural semantics for safe distributed programming

Giuseppe Primiero*
Centre for Logic and Philosophy of Science
Ghent University (Belgium)
IEG - Oxford University
Giuseppe.Primiero@UGent.be

Abstract

In this paper we present a multi-modal polymorphic type system for a computational interpretation of programs with distributed resources. Polymorphism induces a distinction between programs whose code is safe at location, and programs whose value is safe overall. We formulate judgemental modalities to express such distinction and use their introduction and elimination rules to express mobility of code and values within a network. The syntactic formulation is completed by a procedural semantics interpreted over states of an abstract machine for which a standard soundness result is given in the form of a type safety theorem.

Keywords: Modal Type Theory, Distributed Computing, Weak Termination.

1 Introduction

Modalities are becoming common means for the representation of networks embedding diverse procedures, resources, devices, services. In particular, they provide an optimal tool to reason about distributed and staged computation, as proven by the recent large body of work, see e.g. [10, 4, 11, 12, 5]. This is the case especially by endorsing the interpretation of proofs-as-programs made possible by the Curry-Howard isomorphism. In fact, constructive modalities ([2, 1]) have been integrated recently in type theories ([16, 14]) to suggest a new research direction for operational versions of modal logics. In the current literature, we recognize two major approaches to modal logics for distributed computing:

*Fellow of the FWO - Research Foundation Flanders.

1. a syntactic approach, endorsed by the type system presented in [4], where modalities are used to establish global/local temporal validity of propositions, exploiting the notion of contextual accessibility;

2. a semantic approach, like the one put forward by the intuitionistic modal logic $ML5$ for Grid Computing introduced in [11, 12], further enriched with the concept of location.

Other languages to express heterogeneous resources and locations are given for example in [3] and [15].

Varying on the first theme, we present a polymorphic type system with *judgemental* (rather than propositional) modalities: this is a generalization of the system developed in [18] which extends Martin-Löf's Type Theory (MLTT) with derivability from open assumptions. In the present formulation, we introduce multi-modalities to interpret different kinds of procedure termination to reason about safe distributed programming. The use of locally terminating processes to study distributed programming is the major novelty of the present contribution, the most similar approach being presented in [5].

The basic property that characterizes our type system is the underlying polymorphism of constructors, by which a formula including a variable x_i expresses locally valid code by a weakly terminating process, whereas a formula with constant a_i expresses the globally safe value of the corresponding specification. Reduction to normal form will be therefore understood as a procedure of global validation. In turn, this induces a corresponding polymorphism of types depending on the distinction between true and valid assumptions: a type-theoretic judgement $\Gamma_i \vdash A\ \mathtt{type_{inf}}$ is the claim of bounded validity of a specification A in the context of resources and procedures in Γ at address i, containing at least one true assumption $x_i : A$; a type-theoretic judgement $\Delta_i \vdash A\ \mathtt{type}$ is the claim of global validity of a specification A in the context of resources and procedures in Δ at address i, containing all valid assumptions $a_i : A$. To make full use of such distinctions in the language, we extend it by modal judgements: a modal judgement $\Diamond_i(A\ true)$ refers to valid code for specification A, terminating only at some locations (weak termination); a modal judgement $\Box_i(A\ true)$ refers to everywhere executable, safe code for specification A, independent from additional locations being accessed at run-time by the corresponding program a.

The polymorphism and the resulting modal type system are used therefore to explore reasoning about distributed computing, obtaining Code Mobility Rules from corresponding properties of the modal operators. Significant contributions of this work are:

1. the strong assumption on the polymorphic nature of codes and values, understood as locally and globally valid processes;

2. an alternative formulation and interpretation of (multi-)modalities for safe distributed computing;

3. the underlying operational semantics for the interpretation of distributed programs.

The paper is structured as follows. In §2 the system is introduced, first in §2.1 by referring to the polymorphic type system and then in §2.2 by presenting the extension to multi-modalities and their interpretation. In §2.3 the properties of the resulting non-homogeneous modal language are analyzed, focusing on the rules for code mobility. In §3 we present the syntax of a corresponding programming language, its operational semantics and prove a safety result, standardly in terms of preservation and progress properties. In §4 we refer to further applications of this multi-modal type system and other lines of research.

2 A system for code and safe values

Contextual dependency in MLTT provides all that is needed to express syntactically the notion of truth relative to states typical of modal logics. Standardly, in a dependent judgement of the form $\Gamma \vdash a : A$, the proof a of type A holds under appropriate substitutions of assumptions $[x_1/a_1 : A_1, \ldots, x_n/a_n : A_n]$ in Γ. Substitution of variables with proof terms corresponds computationally to their β-reduction or – under the proofs-as-programs interpretation – to explicit evaluation at run-time of codes for $[A_1, \ldots, A_n]$ from which a program for A depends. The distinction between normalized and non-normalized contexts is usually expressed by referring to true and valid assumptions respectively.

The ultimate task of the type system we shall introduce is to formalize the distinction – crucial for distributed computing – between code that can be evaluated only at some location and code that can be evaluated everywhere in a network. One way of approaching this task is to exploit the difference in derivability form true and valid assumptions. In our type system, true assumptions are understood as admitting non-termination at some location(s), so that failure corresponds to a missing address or wrong access to resources; valid assumptions are understood as everywhere terminating processes. On that basis, we draw a syntactic separation between constructors that induce everywhere valid code (mobile to every address) and locally evaluated code (bounded to address), in a manner similar to what studied in [15]. Locally evaluated code can be executed at run-time when required by a main routine; everywhere valid code produces safe values.[1]

We refer to a set $\mathcal{T}$ built from indexed term constructors $a_i, b_j, \ldots$ and variable constructors $x_i, y_j, \ldots$, each for an appropriate type/specification, with $\mathcal{N} = \{i, j, \ldots\}$ an enumerable set of distinct locations in a network. That a formula containing type A holds under a context Γ means: the program for specification A is executable by accessing the (list of) address(es) appearing in Γ and by executing the related code. The constructor for A expresses (implicitely)

[1] We shall not dwell here specifically into the issue of fault-tolerance, to which further research will be devoted. Logically, our distinction presents some analogy with complete versus partial evaluations, see [6].

if the code is bounded to a location or can be broadcasted. This is further refelected both in the type predicated of A and in the corresponding semantic judgemental form. In particular:

- the formula $a_i : A$ says that program a is executed at address i for specification A. The canonical proof-term a defines the corresponding type of safe value A `type`. We further induce an extension of the language with an appropriate $\Box$ modality: from $\Delta_i \vdash a_i : A$, where every expression in Δ_i is evaluated to normal form, we induce the necessity judgement $\Box_{i \in \mathcal{N}}(A\ true)$. This means that the program contains code with complete instructional informations at run-time, all available at i but valid from *any* other location accessible from i, so that it can be safely broadcasted everywhere.

- the formula $x_i : A$ says that a routine x for specification A is validly executable at – and bounded to – address i by some running program. The proof-variable x_i defines the corresponding type of safe code A $\mathtt{type}_{\mathtt{inf}}$. We further induce an appropriate $\Diamond$ modality: from $\Gamma_i \vdash b_i : B$, where $(x_i : A) \in \Gamma_i$, we induce the possibility judgement $\Diamond_{i \in \mathcal{N}}(B\ true)$.

As shown in detail in [19], the set of formulas derivable in the fragment of our type system with $\Box_i$ corresponds to the set of all intuitionistic propositional tautologies, plus the formulas derivable in a multi-modal version of $S4$; on the other hand, the set of formulas derivable in the fragment of our type system with $\Diamond_i$ corresponds to a non-standard fragment thereof, namely a multi-modal version of contextual KT with appropriate axioms for the $\Diamond$ modality. This equivalence allows further interesting results, such as the embedding of the relevant formats of Distributed and Common Knowledge in this language, in line with the standard results from [9] for asynchronous systems.

Our starting point of view is that judgemental assertion conditions can be translated modally to express information on the internal structure of a running program.

2.1 Type System

In this section we define kinds, terms and rules for expressions of our type system.

Definition 1 (Kinds)**.** *The set* $\mathcal{K} =: \{\mathtt{type}, \mathtt{type}_{\mathtt{inf}}\}$ *contains*

- *the kind* `type` *of all specifications valid by everywhere executable programs, defined by term constructors* $\mathcal{C}$*;*

- *the kind* $\mathtt{type}_{\mathtt{inf}}$ *of specifications valid by locally executable codes, defined by variable constructors* $\mathcal{V}$*.*

Definition 2 (Terms)**.** *The set of terms* $\mathcal{T} = \{\mathcal{C}, \mathcal{V}\}$ *is given by:*

- *constructors* $\mathcal{C} := \{a_i; (a_i, b_j); a_i(b_j); \lambda(a_i(b_j)); < a_i, b_j >\}$;
- *variables* $\mathcal{V} := \{x_i; (x_i(b_j)); (x_i(b_j))(a_i)\}$.

An expression $a_i : A$ is the declaration of a safe value for specification A generated by a program running effectively at location i and valid at any other accessible address; an expression $x_i : A$ is a declaration of validity at location i of an instruction to execute a program for A, generating valid code bounded to that address.[2] In turn, a term (a_i, b_j) is a pair of safe values; $a_i(b_j)$ an application (composition) thereof; $\lambda(a_i(b_j))$ produces the value of b_j as a function of a_i; $< a_i, b_j >$ is the ordered pair of those values. Constructions for safe code are used to produce functional constructions only (abstraction and application).

Judgements are generalised to their contextual form:

$$\Gamma_i := \cdot, x_i : A_i, \Delta_i$$
$$\Delta_i := \cdot, a_i : A_i$$

A context Γ_i is built from a finite sequence of typed variables $[x_1 : A_1, \ldots, x_i : A_i]$ in $\mathcal{V}$, all with distinct subjects in $\mathtt{type}_{\mathtt{inf}}$. These formulas express the *locally* executable resources of a network, called upon execution of a program. A judgement $[x_1/a_1 : A_1, \ldots, x_i/a_i : A_i] \vdash A$ `type` says that a value for A is extracted provided each of the routines $A_1, \ldots, A_i$ is called upon at the appropriate location, executed and its value provided. We consider the standard constraint that requires each element $x_i : A_i \in \Gamma$ to depend on the previous one $x_{i-1} : A_{i-1}$ in the same context: this means that the validity of $\Gamma_i \vdash A$ `type` requires the *ordered* execution of procedures in Γ, or in other words that the communication path connecting the different locations in Γ is unidirectional. Contextual dynamics is crucial to represent extension of networks. If $\Gamma = [x_1 : A_1, \ldots, x_i : A_i]$, an extended context $\Gamma' = [\Gamma, x_{i+1} : A_{i+1}]$ is equivalent to $\Gamma' = [x_1 : A_1, \ldots, x_{i+1} : A_{i+1}]$. When the formulation of a fresh declaration $x_{i+1} : A_{i+1}$ representing a new piece of code is meant to be independent of the order in Γ_i, we use a separator $\Gamma_i \mid x_{i+1} : A_{i+1}$. As we want to keep apart open variables and their closed counterparts, we shall refer to the distinction between true and valid assumptions, using Δ_i to refer to a sequence of closed expressions (evaluated terms, valid assumptions) when they occur in a context.[3] In view of the functional construction of terms in `type` by contextual composition with terms in $\mathtt{type}_{\mathtt{inf}}$, the polymorphism of the language can be seen as expressing respectively a Π type with the former and a Σ type over constructions with the latter.[4]

Definition 3 (Rules for *type*)**.** *The kind* $\mathcal{K} :=$ `type` *is governed by the following rules:*

[2] As our modalities are purely judgemental, $\mathcal{T}$ does not contain modal terms, as it is the case with the systems presented in [13] or [11].

[3] In the following, to reduce confusion between the index attached to the constructor – which expresses location – and the index on types – which distinguishes routines, we shall simply use distinct capital letters $A, \ldots N$ as subjects for the latter.

[4] Notice that introduction rules for quantifiers are formulated below for `type`, but for these a specific interpretation holds.

$$\frac{a_i : A}{A \ \texttt{type}} \ \textit{Type Formation}$$

$$\frac{a_i : A \quad b_j : B}{(a_i, b_j) : A \wedge B} \, I\wedge \qquad \frac{(a_i, b_j) : A \wedge B}{a_i : A} \, E \wedge (l)$$

$$\frac{a_i : A \quad A \ type \vdash b_j : B}{a_i(b_j) : A \to B} \, I \to \qquad \frac{a_i(b_j) : A \to B \qquad a_i : A}{B \ \texttt{type}} \, E \to$$

$$\frac{a_1 : A, \ldots, a_n : A \qquad b_j : B[a_i : A] \qquad \lambda((a_i(b_j))A, B}{(\forall a_i : A_i)B \ \texttt{type}} \, I\forall$$

$$\frac{a_1 : A, \ldots, a_n : A \qquad b_j : B[a_i : A] \qquad (< a_i, b_j >, A, B)}{(\exists a_i : A)B \ \texttt{type}} \, I\exists$$

$$\frac{}{\Delta_i, a_j : A \vdash A \ \texttt{type}} \ \textit{Global Validity Rule}$$

$$\frac{\Delta_i \vdash B \ \texttt{type} \quad \Delta_i \vdash A \ \texttt{type}}{\Delta_i, a_i : A \vdash B \ \texttt{type}} \ \textit{Weakening}$$

$$\frac{\Delta_i, a_i : A, b_j : B \vdash C \ \texttt{type} \qquad \Delta_i \vdash b_j : B}{\Delta_i, a_i : A \vdash C \ \texttt{type}} \ \textit{Contraction}$$

$$\frac{\Delta_i \mid a_i : A, b_j : B \vdash C \ \texttt{type}}{\Delta_i \mid b_j : B, a_i : A, \vdash C \ \texttt{type}} \ \textit{Exchange}$$

These construction rules with signed terms a_i, b_j (only) express execution of programs at run-time and their possible composition. Dependency (which usually corresponds to functional abstraction) reduces in this fragment to application for program composition, i.e. composition of functionally executed code. The unusual formulation of the quantifiers is explained as follows: generalization by $\forall$-introduction is an abstraction function on a set of equivalent programs ($a_1, \ldots, a_n$, all of the *same* type A) to extract one instance a_i to be composed by application with another running program; specification by $\exists$-introduction works as a choice function on a similar set of values to pick one value to compose it in a new value: both are hence restricted to enumerable constructors to express quantification over codes. The Global Validity Rule uses premise generation to say that if the safe value of a program for A obtains at address j, its validity is global to the relevant network $\mathcal{N}$ accessible from i. This property makes it possible to validate the other structural rules for expressing modularity: Weakening for explicit formulation of programs running in the same network; Contraction for reduction of different instances of running code; Exchange for irrelevance of ordering on *independent* codes.

Definition 4 (Rules for $\mathtt{type_{inf}}$)**.** *The kind* $\mathcal{K} := \mathtt{type_{inf}}$ *is governed by the following rules:*

$$\frac{\Delta_i \vdash \neg(A \to \bot)\ \mathtt{type} \qquad x_i : A}{A\ \mathtt{type_{inf}}}\ \mathit{Type_{inf}\ Formation}$$

$$\frac{A\ \mathtt{type_{inf}} \qquad x_i : A \vdash b_j : B}{((x_i)b_j) : A \supset B\ \mathtt{type_{inf}}}\ \mathit{Functional\ abstraction}$$

$$\frac{A\ \mathtt{type_{inf}} \qquad x_i : A \vdash b_j : B \qquad a_i : A}{(x(b_j))(a_i) = b_j[x_i/a_i] : B\ \mathtt{type}[x_i/a_i]}\ \beta\text{-}\mathit{conversion}$$

$$\frac{\lambda(a_1(b_j))A, B \qquad (b_j)[a_i := a_j]}{(a_j(b_j)) : A \to B\ \mathtt{type}}\ \alpha\text{-}\mathit{conversion}$$

$$\frac{}{\Gamma_i, x_j : A, \Delta_i \vdash A\ \mathtt{type_{inf}}}\ \mathit{Local\ Validity\ Rule}$$

$$\frac{\Gamma_i \vdash B\ \mathtt{type_{inf}} \qquad x_j : A \vdash A\ \mathtt{type_{inf}}}{\Gamma_i \mid x_j : A \vdash B\ \mathtt{type_{inf}}}\ \mathit{Weakening}$$

$$\frac{\Gamma_i \mid x_j : A, y_j : B \vdash C\ \mathtt{type_{inf}} \qquad \Gamma_i \vdash y_j : B}{\Gamma_i \mid x_j : A \vdash C\ \mathtt{type_{inf}}}\ \mathit{Contraction}$$

$$\frac{\Gamma_i \mid x_j : A \mid y_j : B \vdash C\ \mathtt{type_{inf}}}{\Gamma_i \mid y_j : B \mid x_j : A, \vdash C\ \mathtt{type_{inf}}}\ \mathit{Exchange}$$

Negation introduction, admissible by type-checking on the enumerable constructions, expresses a safety constraint on a running specification in a network: hence the formation rule states legality of code for A in a network, whenever none of its safety constraints fails. Functional abstraction interprets composition of valid code at its location with some executable to produce dependently valid code. The β-conversion rule expresses reduction of the latter to safe value (computationally, reduction to $\mathtt{type}$); α-conversion expresses substitution, by the obvious omitted inductive definition, of an instance of a value constructor in place of a signed variable on a finite domain of equivalent constructors for a class of dependent codes. The Local Validity Rule says that the execution of a program for A at address j bounds A to that location in network $\mathcal{N}$ accessible from i (until discharged by β-conversion, then it becomes safe value everywhere executable). Notice that validity of the structural rules is restricted to assumptions that are not in a relation order within a context: this ensures that order of command execution within a network Γ_i (where at least one expression is of the form $\mathtt{type_{inf}}$), and hence validity of the network itself, is not broken by execution of additional code.

Definition 5 (Semantic Judgements)**.** *The kinding $\mathcal{K}$ induces truth definitions as follows:*

$$\frac{\Delta_i \vdash a_i : A}{A\ true}\ \textit{Global Truth} \qquad \frac{\Gamma_i, \Delta_j \vdash x_i : A}{A\ true^*}\ \textit{Local Truth}$$

A judgement $A\ true$ expresses validity for a specification A by program a generated at address i, without any additional requirement on the network; a judgement $A\ true^*$ says that a specification A is executable and bounded at location i within a network (even when extended to safe values at j).

2.2 Multi-Modalities

Modalities are introduced to express address-boundedness and broadcastable code over a network.

Definition 6 (Modal Judgements)**.** *The set of modal judgements $\mathcal{M}$ for any $i \in \mathcal{N}$ is defined by the following modal formation rules:*

$$\frac{a_i : A}{\Box_i(A\ true)}\ \Box\textit{-Formation} \qquad \frac{x_i : A}{\Diamond_i(A\ true)}\ \Diamond\textit{-Formation}$$

The first rule says that a safe value for A generated at i is admissible at any network extension from i; the second rule says that a valid code for A generated at i is admissible only at some network extension preserving address i; in particular, such network will have to preserve the context Γ_{i-1} of which $x_i : A$ is a valid extension. Context extension mimics accessibility on worlds from standard modal logic and we can express it by modal contexts. Remember that context Γ_i is a context signed for i iff any declaration in Γ has signature i and all have distinct subjects $\{A, \ldots, N\} \in \texttt{type}_{\texttt{inf}}$.

Definition 7 (Modal Contexts)**.** *For any context with true assumptions $\Gamma_i := \{\cdot; x_i : A\}$ and valid assumption $\Delta_j := \{\cdot; a_j : A\}$, we define their modal counterpart as $\Diamond_{i,j\in\mathcal{N}}\Sigma =: \{\Gamma_i, \Delta_j\}$ and $\Box_{i,j\in\mathcal{N}}\Sigma =: \{\cdot; a_i : A, \Delta_j\}$. Where locations $i, j \in \mathcal{N}$ can be safely omitted, we shall abbreviate modal contextual operators to $\Diamond_{\mathcal{N}}, \Box_{\mathcal{N}}$.*

By each distinctly signed (modal) context $\circ_{\mathcal{N}}\Sigma$ (with $\circ = \{\Box, \Diamond\}$), an appropriate modal derivability relation is induced:

Definition 8 (Modal Derivability)**.** *Modal judgements derivable from multi-signed contexts are defined as follows:*

- $\Box_k(A\ true)$ *iff* $\emptyset \mid \Box_{\mathcal{N}}\Sigma \vdash A\ true$, *where* $\mathcal{N} = \{1, \ldots, k-1\}$;
- $\Diamond_k(A\ true)$ *iff* $\Diamond_{\mathcal{N}}\Sigma \vdash A\ true^*$, *where* $\mathcal{N} = \{1, \ldots, k-1\}$.

Given this derivability relation, extension of modal contexts by further modal judgements expresses the requirement that a context extension of $\Box_i\Sigma$ by $\Diamond_j(A\ true)$ is admissible if $\Box_i\Sigma \nvdash (A \to \bot)$.

Introduction and elimination rules for formulas $\Box_{\mathcal{N}}\Sigma \vdash \Box_{\mathcal{N}}(A\ true)$ and $\Diamond_{\mathcal{N}}\Sigma \vdash \Diamond_{\mathcal{N}}(A\ true)$ express global and local validity of $(A\ true)$ in view of code executed at locations $i, j, k \in \mathcal{N}$. To this aim, we now extend the language allowing multiple indices on modal operators: $\Box_{i,j}$ expresses validity of the judgement that follows in the Network accessible from either index; $\Diamond_{i,j}$ expresses validity of the judgement that follows in the Network accessible from both indices.

$$\frac{\Delta_i \mid x_j : A \vdash A\ true^* \qquad \Box_i\Delta, [x_j/a_j] : A \vdash A\ true}{\Box_{\mathcal{N}}\Sigma \vdash \Box_{\mathcal{N}}(A\ true)}\ \text{multiple}\ I\Box$$

$$\frac{\Box_i\Delta, a_j : A \vdash \Box_{i,j}(A\ true) \qquad \Box_{i,j}(A\ true), \Box_k\Delta' \vdash \Box_{i,j,k}(B\ true)}{\Delta_i, a_j : A, \Delta'_k \vdash B\ true}\ \text{multiple}\ E\Box$$

The introduction rule for $\Box$ says that if the only location-bounded code for the execution of a program for A can be validated elsewhere within network Σ, then A can be executed everywhere in Σ. The corresponding elimination starts from a similarly derived $\Box_{\mathcal{N}}(B\ true)$ to decompose its locations: it induces an operational interpretation of a remote procedure call (RPC) presented in extensive format by sending the expression $(A\ true)$ from i, j to $\mathcal{N}$ where it can be used to evaluate B by accessing k.

$$\frac{\Gamma_i \mid x_j : A \vdash B\ true^*}{\Diamond_{i,j}\Sigma \vdash \Diamond_{i,j}(B\ true)}\ \text{multiple}\ I\Diamond$$

$$\frac{\Diamond_i\Gamma, \Box_j\Delta \vdash \Diamond_{i,j}(A\ true) \qquad \Box_j\Delta, x_k : A \vdash \Diamond_{j,k}(B\ true)}{\Gamma_i \mid \Delta_j \vdash B\ true^*}\ \text{multiple}\ E\Diamond$$

The introduction rule for $\Diamond$ says that if execution at i of a program for B requires code bounded to address j, then resources at the intersection of i, j are needed for any execution. This introduction rule construct a return value that can in turn be used for RPC, i.e. a value for B executed at i, j. The corresponding elimination starts from a similarly derivable judgement $\Diamond_{i,j}(A\ true)$ to infer its variable constructor, deriving local validity of B first with and then without the additional location of A (starting from the result of a return value it can be used to give further conditions for RPC).

Lemma 1 (Local Soundness and Completeness of Modal Rules)**.** *The modal rules for $\Box_{\mathcal{N}}, \Diamond_{\mathcal{N}}$ are locally sound and complete.*

Proof. Proof is by by reductions and expansions on modal judgements.

$$\dfrac{\dfrac{\dfrac{D_1}{\Delta_i, a_j : A \vdash A\ true}}{\Box_{i,j}\Sigma \vdash \Box_{i,j}(A\ true)}\Box I \qquad \dfrac{E}{a_j : A, \Delta'_k \vdash B\ true}}{\dfrac{\Delta_i, a_j : A, \Delta'_k \vdash B\ true}{\Delta_i, a_j : A, \Delta'_k \vdash B\ true}}\Box E \Rightarrow_{Redex}$$

$$\dfrac{\dfrac{D_2}{\Delta_i, \Delta'_k \vdash A\ true} \qquad \dfrac{\dfrac{\dfrac{D_1}{\Delta_i, \Delta'_k \vdash A\ true}\Rightarrow_{Exp}}{\Delta_i, a_j : A, \Delta'_k \vdash A\ true}\text{ Global Truth}}{\Box_i\Delta, \Box_j(A\ true), \Box_k\Delta' \vdash \Box_{i,j,k}(A\ true)}\Box I}{\Delta_i, \Delta'_k \vdash A\ true}\Box E$$

$$\dfrac{\dfrac{\dfrac{D_1}{\Gamma_i, x_j : A \vdash B\ true^*}}{\Diamond_{i,j}\Sigma \vdash \Diamond_{i,j}(B\ true)}\Diamond I \qquad \dfrac{E}{\Gamma_i, \Delta_k \vdash A\ true^*}}{\dfrac{\Gamma_i, x_j : A, \Delta_k \vdash B\ true^*}{\Gamma_i, x_j : A, \Delta_k \vdash B\ true^*}}\Diamond E \Rightarrow_{Redex}$$

$$\dfrac{\dfrac{D_2}{\Gamma_i, \Delta_k \vdash A\ true^*} \qquad \dfrac{\dfrac{\dfrac{D_1}{\Gamma_i, \Delta_k \vdash A\ true^*}\Rightarrow_{Exp}}{\Gamma_i, x_j : A, \Delta_k \vdash A\ true^*}\text{ Local Truth}}{\Diamond_i\Gamma, \Diamond_j(A\ true), \Box_k\Delta \vdash \Diamond_{i,j,k}(A\ true)}\Diamond I}{\Gamma_i, \Delta_k \vdash A\ true^*}\Diamond E$$

□

Lemma 2 (Substitutions)**.** *The following substitutions hold:*

1. *If* $\Gamma_i \mid x_j : A, \Delta_k \vdash B\ true^*$ *and* $\Gamma_i, \Delta_k \vdash a_j : A$*, then* $\Gamma_i, \Delta_k \vdash [x_j/a_j]B\ true$*;*
2. *If* $\Box_i\Delta, \Diamond_j(A\ true), \Box_k\Delta' \vdash \Diamond_{i,j,k}(B\ true)$ *and* $\Box_i\Delta, \Box_k\Delta' \vdash \Box_{i,k}(A\ true)$*, then* $\Box_i\Delta, \Box_k\Delta' \vdash \Box_{i,k}(B\ true)$*.*

Proof. Proof is by induction on the length of derivations as follows:

1. On $D_1 = \Gamma_i \mid x_j : A, \Delta_k \vdash B\ true^*$ using Local Truth on $x_j : A$ and using β-conversion on $D_2 = \Gamma_i, \Delta_k \vdash [x_j/a_j] : A$ to infer B `type` and so $B\ true$ by Global Truth.

2. On $D_1 = \Box_i\Delta, \Diamond_j(A\ true), \Box_k\Delta' \vdash \Diamond_{i,j,k}(B\ true)$: from the second premise obtain $x_j : A$ by $E\Diamond$ and $A\ true^*$ by Local Truth; use β-conversion with $[x_j/a_j] : A$ with constant obtained by $\Box_j(A\ true)$ by $E\Box$, followed by $I\Box$. By another instance of $I\Box$ on D_1 obtain $\Box_{i,k}(B\ true)$.

□

2.3 Properties and Code Mobility

In this section we focus on the code mobility rules for the modal type system.

The rule of $\Diamond$-Formation from §2.2 expresses the inference from legality of code for A at i to execution of A for a program in the same network accessible from i. This allows to induce a *Reflexivity* property, satisfied in the simplest case of a single node network. Admitting $\mathcal{N} = \{1, \ldots, n\}, n > 1$, computing is understood as executed in a strictly ordered way, which enforces *Transmission* (or downward only transitivity): if a process for $(C\ true)$ at k takes the computational information expressed by $(B\ true)$ at j, and $(B\ true)$ uses information expressed by $(A\ true^*)$ at i, then the process at k also uses $(A\ true^*)$ at i (for $(i < j < k \in \mathcal{N})$). *Symmetry* is not admitted, as unidirectional communication mimics the fact that routines for a program are processed according to a logical order.[5]

$$\frac{x_i : A \vdash A\ true^*}{x_i : A, \Delta_k \vdash \Diamond_i(A\ true)}\ \text{Reflexivity}$$

$$\frac{x_i : A \vdash A\ true^* \qquad \Diamond_i(A\ true) \vdash \Diamond_j(B\ true) \qquad \Diamond_j(B\ true) \vdash \Diamond_k(C\ true)}{\Diamond_i(A\ true), \Diamond_j(B\ true) \vdash \Diamond_k(C\ true)}\ \text{Transmission}$$

Broadcasting is used to send an execution command for code valid everywhere on the given network to any given intersection with a specific address. *Global Access* is the reverse function: it calls a command from one address j in the network Δ_i to execute program $\Box_{i,j}(B\ true)$. *Convergence* enforces transmission of routines to new accessible locations: if there is a program executed at i which is called upon at j (for the usual $i < j$), then the information used at i is executable at j. Derivability under $\Box_{\mathcal{N}}\Sigma$ allows admissibility of $\Box_k(A\ true)$ by any $\Delta_i, \Delta'_j \in \Box_{\mathcal{N}}\Sigma$ and $i < j < k \in \mathcal{N}$, from which *Upper Inclusion* follows: if a program is actually executed in a network, then it can be accessed from any higher location within that network; *Lower Inclusion* expresses accessibility of executed programs at any lower admissible location in $\mathcal{N}$ (accessibility of the

[5] Failure of symmetry is due to admitting synchronization operations and mobility of safe (evaluated) code in ordered networks. This also means one cannot grant a rule for remote computation as *get* in [12] for $ML5$ ($IS5$), which allows to transfer control and data to reason across (any) worlds, whereas all other rules act locally. Working with everywhere and somewhere evaluated code allows typing-rules for non-local operations but the imposed ordering restricts symmetry. Ordered commands are exemplified in an easy example for compiling a TeX source, DVI outputting and printing in [3].

lower point considered is satisfied by Convergence). By the multi-modal version of $\Box_{1,2}$-Formation, we propagate evaluation on safe contexts: by *Ascending Iteration*, one can access at k a program for A executed at i, j whenever A can be executed at k using processes at i, j; by *Descending Iteration*, one can access at k a program for A executed at i, j, whenever a program for A is executable at k with processes at i, j (a sort of code mobility without Seriality, see [13]). This is easily derivable from Convergence and β-reduction.

$$\frac{\Box_i\Delta, a_j : A \vdash \Box_{i,j}(B\ true) \qquad x_j : A \vdash A\ true^*}{\Box_i\Delta, \Diamond_j(A\ true) \vdash \Diamond_{i,j}(B\ true)}\ \text{Broadcasting}$$

$$\frac{\Delta_i, x_j : A \vdash B\ true^* \qquad a_j : A \vdash A\ true}{\Box_i\Delta, a_j : A \vdash \Box_{i,j}(B\ true)}\ \text{Global Access}$$

$$\frac{\Box_i\Delta \vdash A\ true \qquad x_i : A \vdash \Diamond_j(A\ true)}{\Box_i\Delta, x_i : A \vdash \Diamond_j(A\ true)}\ \text{Convergence}$$

$$\frac{\Box_i\Delta, \Box_j\Delta' \vdash \Box_k(A\ true) \qquad \Box_{i,j}\Sigma \mid a_k : A \vdash \Box_{i,j,k}(A\ true)}{\Box_{i,j}\Sigma \vdash \Box_{i,j}(A\ true)}\ \text{Upper Inclusion}$$

$$\frac{\Box_i\Delta, \Box_j\Delta' \vdash \Box_{i,j}(A\ true) \qquad \Box_{i,j}\Sigma \vdash \Box_k(A\ true)}{\Box_{i,j}\Sigma \vdash \Box_k(A\ true)}\ \text{Lower Inclusion}$$

$$\frac{\Box_i\Delta, \Box_j\Delta' \vdash \Box_k(A\ true)}{\Box_{i,j}\Sigma \vdash \Box_k(\Box_{i,j}(A\ true))}\ \text{Ascending Iteration}$$

$$\frac{\Box_i\Delta, \Box_j\Delta' \vdash \Box_k(A\ true)}{\Box_{i,j}\Sigma \vdash \Box_{i,j}(\Box_k(A\ true))}\ \text{Descending Iteration}$$

3 Abstract Syntax and Procedural Semantics

In this section we consider the abstract syntax underlying our type system and an operational semantics that defines syntactic transformations on states of the language, i.e. a sequence of evaluations of machine configurations defined by rewriting rules. We conclude by showing under which conditions safe code is expressed, obtaining standard Safety and Preservation results on computations.

Expressions of the language are composed from the syntax in terms of a ternary relation $\mapsto (\Gamma_i, t_i)$ out of a context Γ_i, a well-typed term t_i and an evaluation function $\mapsto$. The context is a stack (possibly a singleton) of well-typed terms that are either globally evaluated (values) or locally evaluated (code); it

maps identifiers to (possibly local) values and function definitions; the distinction between global and local evaluations is preserved by the modalities.[6] The term in an expression is accordingly typed either as an everywhere evaluable term or as a locally evaluable term by appropriate functions (respectively, *exec* and *run*). Evaluation results from their contextual version coupled with mobility functions ($GLOB$ and $BROAD$ for Remote Procedure Calls; RET and SEN for Portable Code). To preserve the order of commands we add ordered intersection $i \cap j$ and union $i \cup j$ at locations: the former expresses the requirement that the function to which is applied be executed orderly on the first and second index of the intersection; the latter expresses the validity of the related function on the network accessible from either of the indices of the union.[7]

Definition 9 (Syntax)**.** *The syntax is defined by the following alphabet:*

$\texttt{Types} := \{\alpha \mid \alpha \times \beta \mid \alpha + \beta \mid \alpha \to \beta \mid \alpha \supset \beta\}$

$\texttt{Terms} := \{x_i \mid a_i, \textit{ for } i \in \texttt{Indices}\}$

$\texttt{Indices} := \{1, \ldots, n\}$

$\texttt{Functions} := \{exec(\alpha) \mid run_i(\alpha) \mid run_{i \cup j}(\alpha \cdot \beta) \mid run_{i \cap j}(\alpha \cdot \beta) \mid synchro_j(\beta(exec(\alpha)))\}$, *where* $\cdot = \{+, \times\}$

$\texttt{Contexts} := \{\Gamma_i \mid \circ_i \Gamma\}$, *where* $\circ = \{\Box, \Diamond\}$

$\texttt{Remote Operations} := \{GLOB(\Box_{i \cup j}\Gamma, \alpha) \mid BROAD(\Diamond_{i \cap j}\Gamma, \alpha)\}$

$\texttt{Portable Code} := \{RET(\Gamma_{i \cup j}, \alpha) \mid SEND(\Gamma_{i \cap j}, \alpha)\}$

Syntactic expressions are then evaluated in a model defined by states of the machine.

Definition 10 (Operational Model)**.** *The set* $\texttt{States} := \{S, S', \ldots\}$ *contains states of the machine. A state*

$$S := (\mathcal{C}, t.i : \alpha) \mid \mathcal{C} \in \texttt{Contexts}; t \in \texttt{Terms}; i \in \texttt{Indices}; \alpha \in \texttt{Types}$$

is an occurrence of an indexed typed term in context. An operational model of the procedural semantics for the machine is a model where each S *is evaluated by transition to some* S'*. An indexed transition system, called a* `Network`

$$\texttt{Network} := (\mathcal{S}, \mapsto, \mathcal{I})$$

is a triple with $\mathcal{S} \subseteq \texttt{States}$, $\mathcal{I} \subseteq \texttt{Indices}$ *and* $\mapsto$ *a ternary relation over indexed states* $(\mathcal{S} \times \mathcal{I} \times \mathcal{S})$*. If* $S, S' \in \mathcal{S}$ *and* $i, j \in \mathcal{I}$*, then* $\mapsto (S, i, j, S')$ *is written as* $S_i \mapsto S'_j$*. This means that there is a transition* $\mapsto$ *from state* S *valid at index* i *to state* S' *valid at index* j *defined according to the machine typing rules.*

[6] We avoid here to burden further the notion with distinct letters for set of values and sets of code terms and entirely rely on modal contexts to express this distinction.

[7] This additional requirement on the indices replaces the strict ordering on contexts formulated for the type theory.

The procedural semantics expresses evaluation of typed expressions in states by reduction to a terminal one. Formulas in a terminal state produce an otuput value (valid formulas). In the following we list the rewrite operations from state to state of the machine.

Definition 11 (Network State)**.** *The rewriting of a state machine S into another state machine S' is established by the following rules:*

	$S \mapsto S'$
`run`	$(\Gamma_i, x_i : \alpha) \mapsto (\Diamond_i \Gamma, run_i(\alpha))$
`exec`	$(\Gamma_i, a_i : \alpha) \mapsto (\Box_i \Gamma, exec(\alpha))$
`corun`	$(\Gamma_i, run_i(\alpha) \vdash b_j : \beta) \mapsto (\Box_i \Gamma, run_{i \cap j}(\alpha(\beta))$
`coexec`	$(\Gamma_i, exec(\alpha) \vdash b_j : \beta) \mapsto (\Box_i \Gamma, run_{i \cup j}(\alpha(\beta))$
`synchro`	$(\Box_i \Gamma, run_{i \cup j}(\alpha(\beta)) \mapsto (\Box_i \Gamma, synchro_j(\beta(exec(\alpha))))$
`product`	$(\Gamma_i, exec(\alpha), exec(\beta)) \mapsto (\Box_i \Gamma, run_{i \cap j}(\alpha \times \beta))$
`extraction1`	$(\Box_i \Gamma, run_{i \cap j}(\alpha \times \beta)) \mapsto (\Box_i \Gamma, exec(\alpha))$
`extraction2`	$(\Box_i \Gamma, run_{i \cap j}(\alpha \times \beta)) \mapsto (\Box_i \Gamma, exec(\beta))$
`tagunion`	$(\Gamma_i, exec(\alpha)) \mapsto (\Box_i \Gamma, run_{i \cup j}(\alpha + \beta))$
`patternmatch1`	$(\Box_i \Gamma, run_{i \cup j}(\alpha + \beta) \vdash c_k : \gamma) \mapsto (\Box_i \Gamma, run_{i \cap k}(\alpha(\gamma)))$
`patternmatch2`	$(\Box_i \Gamma, run_{i \cup j}(\alpha + \beta) \vdash c_k : \gamma) \mapsto (\Box_i \Gamma, run_{j \cap k}(\beta(\gamma)))$
`□1`	$(\Box_i \Gamma, exec(\alpha)) \mapsto (GLOB(\Box_{i \cup j} \Gamma, \alpha))$
`□2`	$(\Box_{i \cup j} \Gamma, \alpha) \mapsto (RET(\Gamma_{i \cup j}, \alpha))$
`◇1`	$(\Diamond_i \Gamma, run_j(\alpha)) \mapsto (BROAD(\Diamond_{i \cap j} \Gamma, \alpha))$
`◇2`	$(\Diamond_{i \cap j} \Gamma, \alpha) \mapsto (SEND(\Gamma_{i \cap j}, \alpha))$

The rewriting rules are defined by typing rules of an analytic proof system:

Definition 12 (Typing Rules)**.** *The set of typing rules is:*

$$\frac{}{\Delta_i, a_i : \alpha \vdash exec(\alpha)}\ Global \qquad \frac{}{\Gamma_i, x_i : \alpha; \Delta_i \vdash run_i(\alpha)}\ Local$$

$$\frac{a_i : \alpha \qquad b_j : \beta}{run_{i \cap j}(\alpha \times \beta)}\ I\times \qquad \frac{run_{i \cap j}(\alpha \times \beta)}{exec(\alpha)}\ E \times (l)$$

$$\frac{a_i : \alpha}{run_i(\alpha + \beta)}\ I + (1) \qquad \frac{b_j : \beta}{run_j(\alpha + \beta)}\ I + (2)$$

$$\frac{run_{i \cup j}(\alpha + \beta) \qquad run_i(\alpha) \vdash c_k : \gamma \qquad run_j(\beta) \vdash c_k : \gamma}{run_{i \cap k; j \cap k}(\gamma)}\ E+$$

$$\frac{x_i : \alpha \qquad run_i(\alpha) \vdash b_j : \beta}{run_{i \cap j}(\alpha \supset \beta)}\ I \supset \qquad \frac{a_i : \alpha \qquad exec(\alpha) \vdash b_j : \beta}{run_{i \cup j}(\alpha \to \beta)}\ I \to$$

$$\frac{run_{i \cap j}(\alpha \supset \beta) \qquad a_i : \alpha}{synchro_j(\beta(exec(\alpha)))}\ Synchro$$

$$\frac{\Gamma_i, x_j : \alpha \vdash run_j(\alpha) \qquad \Box_i\Gamma, x_j(a_j) : \alpha \vdash exec(\alpha)}{GLOB(\Box_{i\cup j}\Gamma, \alpha)}\ RPC1$$

$$\frac{\Gamma_i, x_j : \alpha \vdash run_j(\alpha) \qquad \Diamond_i\Gamma \vdash run_j(\alpha)}{BROAD(\Diamond_{i\cap j}\Gamma, \alpha)}\ RPC2$$

$$\frac{\Box_i\Gamma, a_j : \alpha \vdash exec(\alpha) \qquad GLOB(\Box_{i\cup j}\Gamma, \alpha)}{RET(\Gamma_{i\cup j}, \alpha)}\ PORT1$$

$$\frac{\Box_i\Gamma, x_j : \alpha \vdash run_{i\cap j}(\alpha) \qquad BROAD(\Diamond_{i\cap j}\Gamma, \alpha)}{SEND(\Gamma_{i\cap j}, \alpha)}\ PORT2$$

The semantics of these rules, as interpreted by state-rewriting, is composed by non-terminal and terminal states. The former are those states that always admit a further mapping to another state of the machine. A terminal state corresponds to a state containing a typed expression in normal form.

Definition 13 (Semantic Expressions)**.** *Evaluation defines strong typing (normalisation) by reduction to states obtained by rules* `exec`*,* `coexec` *and* $\Box$`1` *in the last step of rewriting. Evaluation defines weak typing by expressions in states obtained by rewriting rules* `run`*,* `corun` *and* $\Diamond$`1`*, which give admissible procedural steps but may fail to produce a safe value when a wrong address is called upon by the next state. Normal transitions are those outputting formulas of the form* $\Box_i\Gamma, exec(\alpha)$ *as values.*

Transitions according to `run`, `corun`, `synchro`, $\Diamond$`1/2` all require index preservation. The evaluation on contexts proceeds on the ordering induced by $i < j$. A transition $S \mapsto S'$ consists of

- decomposing a state S into an evaluation context (if present) and an instruction;
- the evaluation of the context and the execution of the instruction;
- the replacement of instruction execution in one of the rules to obtain S'.

Standardly, our operational semantics allows to establish safety for the type system by telling how the program executes. Our notion of safety is constrained by normalization to values as by Definition 13. We use the standard way to prove safety by showing progress and preservation. By progress, one shows that if a program is well-typed, either at the last step reached there is another state that follows, or this provides a semantic value as by Definition 13. By preservation, one shows that if a program $t.i$ is well-typed in α and it reduces to $t.j$, then the latter is also typed in α (or its type can be obtained by appropriate rewriting from α).[8]

[8]Standard reference for these results is [17].

Theorem 1 (Progress)**.** *If* $S := (\Gamma, t.i : \alpha)$*, then either* $S \mapsto S'$ *or* $exec(\alpha)$ *is the output value.*

Proof. By induction on the last rewriting step to obtain $t.i : \alpha$.

1. $(\Gamma_i, a_i : \alpha) \mapsto (\Box_i \Gamma, exec(\alpha))$ by `exec`, and so $exec(\alpha)$ is the defined value;
2. $(\Gamma_i, exec(\alpha)) \mapsto (\Box_i \Gamma, \alpha)$ by $\Box 1$ and $exec(\alpha)$ is the defined value;
3. $(\Gamma_i, x_i : \alpha) \mapsto (\Diamond_i \Gamma, run_i(\alpha))$ by `run`, then the following subcases apply:
   - 3.1 for all $i < j$: $(\Gamma_i, run(\alpha)) \mapsto (BROAD(\Diamond_{i \cap j} \Gamma, \alpha))$; then $SEND(\Gamma_{i \cap j}, \alpha)$ by $\Diamond 2$ and for each such i, j $(\Gamma_i, a_i : \alpha) \mapsto (\Box_i \Gamma, exec(\alpha))$, then proceeds as by Case 1.
   - 3.2 there is no such j, then immediately apply `exec` to obtain $exec(\alpha)$ as value, as by Case 1.
   - 3.3 for some $i < j$: $\Gamma_j, run(\alpha \supset \bot)$; proceed by 4.1 with $\neg\alpha$.
4. $(\Gamma_i, run_i(\alpha) \vdash b_j) \mapsto (\Box_i \Gamma, run_{i \cap j}(\alpha(\beta)))$ by `corun`: it requires the sequential processing of $run_i(\alpha) \vdash b_j$:
   - 4.1 if either of 3.1 and 3.2 apply, obtain $exec(\alpha)$; then by `synchro` obtain the parallel processing of $exec(\alpha) \vdash exec(\beta)$, hence define the value.
   - 4.2 if 3.3 applies, proceed by 4.1 with $\neg\alpha$.
5. $(\Gamma_i, exec(\alpha) \vdash b_j) \mapsto (\Box_i \Gamma, run_{i \cup j}(\alpha(\beta)))$ by `coexec`: reduces to 4.1 with the first step already performed, hence apply `synchro` and define the value.
6. $(\Gamma_i, exec(\alpha), exec(\beta)) \mapsto (\Box_i \Gamma, run_{i \cap j}(\alpha \times \beta))$ by `product`: apply sequentially `extraction1` and `extraction2`, define the value.
7. $(\Gamma_i, exec(\alpha)) \mapsto (\Box_i \Gamma, run_{i \cup j}(\alpha + \beta))$ by `tagunion`:
   - 7.1 apply `patternmatch1` to obtain $(\Box_i \Gamma, run_{i \cap k}(\alpha(\gamma)))$: apply `corun` at 4, if 4.1 is successful, value is obtained; otherwise go to next step.
   - 7.2 apply `patternmatch2` to obtain $(\Box_i \Gamma, run_{j \cap k}(\beta(\gamma)))$: apply `corun`, at 4 and define the value.

□

Theorem 2 (Preservation)**.** *If* $S := (\Gamma, t.i : \alpha)$ *and* $S \mapsto S'$*, then* $S' := (\Gamma, t' : \alpha)$.

Proof. The proof goes by induction on the derivation step of t' and the structure of Γ.

1. The proof holds vacuosly if $t' := exec(\alpha))$, which happens if the bottom rule is one of `exec`, `coexec`, $\Box 1$.

2. $t' := run_i(\alpha)$ is obtained if the bottom rule is one of

   2.1 `run`: then $S := (\cdot, x_i : \alpha)$ and $S' := (\cdot, run_i(\alpha))$, so types are preserved;

   2.2 `corun`: then $S := (run_i(\alpha) \vdash exec(\beta))$ by subderivations of $x_i : \alpha$ and $b_j : \beta$, and $S' := (run_{i \cap j}(\alpha(\beta)))$, so types are preserved;

   2.3 `product`: then $S := (\cdot, exec(\alpha), exec(\beta))$, i.e. by subderivations of $a_i : \alpha; b_j : \beta$ on gets $S' := (\cdot, run_{i \cup j}(\alpha \times \beta))$, so types are preserved;

   2.4 `tagunion`: then $S := (\cdot, exec(\alpha))$ by subderivations of $a_i : \alpha$ and $S' := (\cdot, run_i(\alpha + \beta))$.

3. The proof holds vacuosly for the corresponding analytic rules: `synchro`, `extraction1/2` and `patternmatch1/2`.

4. If the bottom rule is $\Box\mathtt{2}$, then $S := (\Box_{i \cup j}\Gamma, \alpha)$ and $S' := (RET(\Gamma_{i \cup j}, \alpha))$, which is nothing else than an abbreviation for $S'' := (\Gamma_{i \cup j}, run_{i \cup j}(\alpha))$; a step as in 2.2 applies.

5. If the bottom rule is $\Diamond\mathtt{1}$, then $S := (\Diamond_i\Gamma, run_j(\alpha))$ and $S' := (BROAD(\Diamond_{i \cap j}\Gamma, \alpha))$, then the following subcases apply:

   4.1 for all $i < j$, $S'' := (BROAD(\Diamond_{i \cap j}\Gamma, \alpha))$;

   4.2 there is no such j and S'', then halt;

   4.3 for some $i < j$, $S'' := (BROAD(\Diamond_{i \cap j}\Gamma, \neg\alpha))$; proceed by `corun` with $\neg\alpha$.

6. If the bottom rule is $\Diamond\mathtt{2}$, then $S := (\Diamond_{i \cap j}\Gamma, \alpha)$ and $S' := (SEND(\Gamma_{i \cap j}, \alpha))$, which abbreviates $S'' := (\Gamma_{i \cap j}, run_{i \cap j}(\alpha))$; any of the preservation step according to 2. applies.

□

Theorem 3 (Type Safety)**.** *Safety is satisfied by transformations (according to the table in Definition 11) or by terminating expression (exec(α)):*

1. *If $S := (t.i : \alpha)$, and $S \mapsto S'$, then $S' := (t.i : \alpha)$;*
2. *If $S := (t.i : \alpha)$, then either exec(α) is the output value or there is α' for $S' := (t.i : \alpha')$ s.t. $S \mapsto S'$.*

Proof. Part (i) is obtained by Theorem 2; Part (ii) by Theorem 1. □

4 Conclusion

We have presented a modal type system for reasoning about (safe) distributed computing, with polymorphism reflecting weak and strong typing. According to its operational interpretation, typing prevents programs from accessing resources at locations where they are unavailable and rules express where code can be moved. Computationally, values of $\Box_{\mathcal{N}}(A\ true)$ express everywhere accessible code; terms in $\Box_i\Gamma$ refer to computations executed at address i further broadcastable in the network for the execution at runtime of a program. The values of $\Diamond_{\mathcal{N}}(A\ type)$ represent locally accessible code; terms in $\Diamond_i\Gamma$ refer to programs bounded to the specified addresses, up to i. Code sources are ordered as to mimic their functional aspect and modal interaction simulates the validity of code at distinct locations.

Further lines of research are the simulation of open terms in interactive theorem proving, as variables that are intended to be bound but whose binders are not constructed yet (see [8]), and the study of fault-tolerance methods for distributed computing to produce correct results despite fault addressing of resources (for an introductory overview, see [7]).

References

[1] N. Alechina, M. Mendler, V. de Paiva, and E. Ritter. Categorical and Kripke Semantics for Constructive S4 Modal Logic. In *Proceedings of the 15th International Workshop on Computer Science Logic*, volume 2142 of *Lecture Notes In Computer Science*, pages 292 – 307, 2001.

[2] G.M. Bierman and V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, (65):383–416, 2000.

[3] T. Borghuis and L.M.G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, pp.274–289, vol.43, n.4, 2000.

[4] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[5] L. Jia and D. Walker. Modal Proofs as Distributed Programs. In *Programming Languages and Systems, ESOP2004*, volume 2986 of *Lectures Notes in Computer Science*. Springer Verlag, 2004.

[6] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall International, 1993.

[7] F.C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, Vol. 31, No. 1, pp. 1-26, March 1999

[8] H. Geuvers, G. I. Jojgov. Open proofs and open terms: A basis for interactive logic. In Julian C. Bradfield, editor, *Proc. of 16th Int. Wksh. on Computer Science Logic, CSL 2002 (Edinburgh, UK, 22–25 Sept. 2002)*, volume 2471, pages 547–552. Berlin, 2002.

[9] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[10] J. Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 2003.

[11] T. Murphy. *Modal Types for Mobile Code*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2008. CMU-CS-08-126.

[12] T. Murphy, K. Crary, and R. Harper. *Type-Safe Distributed Programming with ML5*, volume 4912 of *Lectures Notes in Computer Science*, pages 108–123. Springer Verlag, 2008.

[13] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In H. Ganzinger, editor, *Proceedings of the 19th Annual Symposium on Logic in Computer Science (LICS'04)*, pages 286–295. IEEE Computer Society Press, 2004.

[14] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–48, 2008.

[15] S. Park. A modal language for the safety of mobile values. In Fourth ASIAN Symposium on Programming Languages and Systems, 2006, pp.217–233, Springer.

[16] F. Pfenning and R. Davies. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[17] B. Pierce. *Types and Programming Languages*, MIT Press, 2002.

[18] G. Primiero. A contextual type theory with judgemental modalities for reasoning from open assumptions. *Logique & Analyse*, vol. 220, 2012 (to appear).

[19] G. Primiero, M. Taddeo. A modal type theory for formalizing trusted communications. *Journal of Applied Logic*, 10, pp.92–114, 2012. DOI: 10.1016/j.jal.2011.12.002.