A Modal Language for the Safety of Mobile Values

Sungwoo Park

Pohang University of Science and Technology, Republic of Korea
gla@postech.ac.kr

Abstract. In the context of distributed computations, local resources give rise to an issue not found in stand-alone computations: the safety of mobile code. One approach to the safety of mobile code is to build a modal type system with the modality $\Box$ that corresponds to necessity of modal logic. We argue that the modality $\Box$ is not expressive enough for safe communications in distributed computations, in particular for the safety of mobile values. We present a modal language which focuses on the safety of mobile values rather than the safety of mobile code. The safety of mobile values is achieved with a new modality $\boxdot$ which expresses that given code evaluates to a mobile value. We demonstrate the use of the modality $\boxdot$ with a communication construct for remote procedure calls.

1 Introduction

A distributed computation is a cooperative process taking place in a network of nodes. Each node is capable of performing a stand-alone computation and also communicating with other nodes to distribute and collect code and data. Thus a distributed computation has the potential to make productive use of all the nodes in the network simultaneously.

Usually a distributed computation assumes a heterogeneous group of nodes with different *local resources*. A local resource can be either a permanent/physical object available at a particular node (*e.g.*, printer, database) or an ephemeral/semantic object created during a stand-alone computation (*e.g.*, heap cell, abstract data type). Local resources are accessed via their references (*e.g.*, handle for a database file, pointer to a heap cell).

Local resources, however, give rise to an issue not found in stand-alone computations: the safety of *mobile code*, or in our terminology, the safety of *mobile terms* where a term represents a piece of code. In essence, a node cannot access remote resources in the same way that it accesses its own local resources, but it may receive mobile terms in which references to remote resources are exposed. Therefore the safety of mobile terms is achieved by supporting direct access to remote resources (*e.g.*, remote file access, remote memory access), as in Obliq [1], by transmitting copies of local resources along with mobile terms, as in Facile [2], by preventing references to remote resources from being dereferenced, as in Mobile UNITY [3], or by allowing all of these methods, as in λdist [4]. Our paper focuses on the third case where we reject mobile terms containing references to remote resources.

One approach to the safety of mobile terms is to build a modal type system with the modality $\Box$ [5,6,7,8] which is based on a spatial interpretation of necessity of modal logic such as S4 and S5. The basic idea is that a value of modal type $\Box A$ contains a

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 217–233, 2006.

mobile term that can be evaluated at any node. By requiring that a mobile term be from a value of type $\Box A$, we ensure its safety without recourse to runtime checks.

A type system augmented with the modality $\Box$ is not, however, expressive enough for safe communications of *values*, *i.e.*, the safety of *mobile values*. In other words, we cannot rely solely on modal types $\Box A$ to verify that a value communicated from one node to another is mobile (*e.g.*, when a remote procedure call returns, or when a value is written to a channel). The reason is that in general, a value of type $\Box A$ contains *not a mobile value but a mobile term*. The evaluation of such a mobile term (with the intention of obtaining a mobile value) may result in a value that is not necessarily mobile because of references to local resources created during the evaluation.

As an example, consider a term of type `int -> int` in an ML-like language:

```
let
  val new_reference = ref 0
  val f = fn x => x + !new_reference
in
  f
end
```

The above term can be evaluated at any node and thus may be used in building a mobile term of type `□(int -> int)`. The resultant value `f`, however, is not mobile because it accesses a local resource `new_reference`. In contrast, the following term, also of type `int -> int`, cannot be used in building a mobile term of type `□(int -> int)`, but the resultant value is mobile because it does not access any local resource:

```
let
  val v = !some_existing_reference
  val f = fn x => x + v
in
  f
end
```

Hence the modality $\Box$ is irrelevant to the safety of mobile values, which should now be verified by programmers themselves.

This paper investigates a new modality $\boxdot$ which expresses that a given term evaluates to a mobile value. The basic idea is that a term contained in a value of modal type $\boxdot A$ evaluates to a value that is valid at any node. For example, the first term above cannot be used in building a term of type `⊡(int -> int)`, but the second term above may be used in building such a term. To obtain a value to be communicated to other nodes, we evaluate a term contained in a value of type $\boxdot A$. In this way, we achieve the safety of mobile values.

While the mobility of a term is independent of the mobility of the value to which it evaluates, the modality $\Box$ is weaker than the modality $\boxdot$ in that we can emulate $\Box$ with $\boxdot$. For example, we may define $\Box A$ as `⊡(unit -> A)`, in which case we check the mobility of a term M of type A by checking the mobility of a value `fn _ =>` M of type `unit -> A`. Thus $\boxdot$ is inherently more expressive than $\Box$, and the use of $\boxdot$ practically eliminates the need for $\Box$. The converse is not the case, however: we cannot

emulate $\boxdot$ with $\Box$ because the modality $\boxdot$ requires at least a distinction between terms and values, which is not provided by the type system for the modality $\Box$.

Since the modality $\Box$ is inadequate for ensuring the safety of mobile values, safe communications are restricted to mobile terms if the underlying type system uses only $\Box$. Such a restriction leads to an unusual implementation of common communication constructs in distributed computations. For example, in λ_{rpc} by Jia and Walker [7] and *Lambda 5* by Murphy *et al.* [8], a remote procedure call returns a mobile term (instead of a mobile value) which the caller node needs to further evaluate in order to obtain the final result of the remote procedure call. By focusing on mobile values rather than mobile terms, the modality $\boxdot$ avoids such anomalies and gives a faithful implementation of common communication constructs.

In Sections 2 and 3, we develop a call-by-value language $\lambda_{\boxdot}$ with mutable references and the modality $\boxdot$. We choose mutable references as a representative example of local resources; other kinds of local resources can be treated in an analogous way. We formulate its type system in the natural deduction style by giving introduction and elimination rules for each connective and modality. The modality $\boxdot$ requires us to introduce a typing judgment differentiating values from terms. The type system takes into account *primitive types* (such as boolean values and integers) for which mobility is an inherent property.

In Section 4, we develop $\lambda_{\boxdot}$ into $\lambda^N_{\boxdot}$ which has a *network operational semantics* and is thus capable of modeling distributed computations. We demonstrate the use of modal types with a communication construct for remote procedure calls. The safety of mobile terms and mobile values is shown by type safety of $\lambda^N_{\boxdot}$, *i.e.*, its progress and type preservation properties.

Section 5 compares $\lambda^N_{\boxdot}$ with other modal languages for distributed computations. Section 6 concludes with future work. Due to space limitations, we refer the reader to our technical report [9] for details of all proofs.

2 Call-by-Value Language λ with Mutable References

This section reviews the type system of λ, a typical call-by-value language with mutable references, in the context of distributed computations. Figure 1 shows the definition of λ.

The syntax of λ uses metavariables A, B, C for types and M, N for terms. () is an expression of type unit, which we include as an example of a primitive type. $\lambda x\!:\!A.\, M$ and $M\ M'$ are a λ-abstraction and a λ-application, respectively. $\mathsf{ref}\ M$ allocates a fresh reference, $!M$ dereferences an existing reference, and $M := M'$ assigns a new value to a reference; a location l, of type $\mathsf{ref}\ A$, is a value for a reference.

A variable x with binding $x : A$ is assumed to hold a value because λ uses the call-by-value strategy. We use a typing judgment $\Gamma \mid \Psi \vdash M : A$ to mean that term M has type A under typing context Γ and store typing Ψ; a store typing judgment $\Psi \vdash \psi\ \mathsf{okay}$ means that store ψ conforms to store typing Ψ.

The operational semantics of λ uses a reduction judgment $M \mid \psi \longrightarrow M' \mid \psi'$ to mean that term M with store ψ reduces to term M' with store ψ' where $\psi = \psi'$ is allowed. A β-reduction judgment $M \longrightarrow_\beta M'$ uses a capture-avoiding substitution $[V/x]M$ defined in a standard way. We write $\phi[\![M]\!]$ for a term obtained by filling the hole [] in an

$$
\begin{array}{lrcl}
\text{type} & A & ::= & \mathsf{unit} \mid A \to A \mid \mathsf{ref}\ A \\
\text{term} & M & ::= & () \mid x \mid \lambda x{:}A.\, M \mid M\ M \mid \mathsf{ref}\ M \mid !M \mid M := M \mid l \\
\text{value} & V & ::= & () \mid \lambda x{:}A.\, M \mid l \\
\text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{store typing} & \Psi & ::= & \cdot \mid \Psi, l \mapsto A \\
\text{store} & \psi & ::= & \cdot \mid \psi, l \mapsto V
\end{array}
$$

$$\frac{}{\Gamma \mid \Psi \vdash () : \mathsf{unit}}\ \mathsf{Unit}$$

$$\frac{x : A \in \Gamma}{\Gamma \mid \Psi \vdash x : A}\ \mathsf{Var} \qquad \frac{\Gamma, x : A \mid \Psi \vdash M : B}{\Gamma \mid \Psi \vdash \lambda x{:}A.\, M : A \to B}\ \to\mathsf{I} \qquad \frac{\Gamma \mid \Psi \vdash M : A \to B \quad \Gamma \mid \Psi \vdash N : A}{\Gamma \mid \Psi \vdash M\ N : B}\ \to\mathsf{E}$$

$$\frac{\Gamma \mid \Psi \vdash M : A}{\Gamma \mid \Psi \vdash \mathsf{ref}\ M : \mathsf{ref}\ A}\ \mathsf{Ref} \qquad \frac{\Gamma \mid \Psi \vdash M : \mathsf{ref}\ A}{\Gamma \mid \Psi \vdash !M : A}\ \mathsf{Deref} \qquad \frac{\Gamma \mid \Psi \vdash M : \mathsf{ref}\ A \quad \Gamma \mid \Psi \vdash N : A}{\Gamma \mid \Psi \vdash M := N : \mathsf{unit}}\ \mathsf{Assign}$$

$$\frac{\Psi(l) = A}{\Gamma \mid \Psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc} \qquad \frac{dom(\Psi) = dom(\psi) \quad \cdot \mid \Psi \vdash \psi(l) : \Psi(l) \ \text{ for every } l \in dom(\psi)}{\Psi \vdash \psi\ \mathsf{okay}}\ \mathsf{Store}$$

$$(\lambda x{:}A.\, M)\ V \quad \longrightarrow_\beta \quad [V/x]M$$

$$\text{evaluation context} \quad \phi \quad ::= \quad [] \mid \phi\ M \mid V\ \phi \mid \mathsf{ref}\ \phi \mid !\phi \mid \phi := M \mid V := \phi$$

$$\frac{M \longrightarrow_\beta M'}{\phi[\![M]\!] \mid \psi \longrightarrow \phi[\![M']\!] \mid \psi}\ Red_\beta \qquad \frac{l \notin dom(\psi)}{\phi[\![\mathsf{ref}\ V]\!] \mid \psi \longrightarrow \phi[\![l]\!] \mid \psi, l \mapsto V}\ Ref$$

$$\frac{\psi(l) = V}{\phi[\![!l]\!] \mid \psi \longrightarrow \phi[\![V]\!] \mid \psi}\ Deref \qquad \frac{}{\phi[\![l := V]\!] \mid \psi \longrightarrow \phi[\![()]\!] \mid [l \mapsto V]\psi}\ Assign$$

Fig. 1. Definition of the language λ

evaluation context ϕ with M. $[l \mapsto V]\psi$ replaces $l \mapsto V'$ in ψ by $l \mapsto V$; we write $\psi(l)$ and $\Psi(l)$ for the value and the type to which l is mapped under ψ and Ψ, respectively.

In the context of distributed computations, $x : A$ in a typing context Γ means that variable x holds a value of type A that is valid at a hypothetical node where typechecking takes place, which we call the *current node* throughout the paper. Then a typing judgment $\Gamma \mid \Psi \vdash M : A$ means that if both typing context Γ and store typing Ψ are satisfied, the evaluation of term M at the current node returns a value V of type A. It does not, however, tell us if M is a mobile term that can be evaluated at other nodes. More importantly, it does not tell us if V is a mobile value that is valid at other nodes. Therefore the above type system is not expressive enough for the safety of mobile terms and mobile values in distributed computations.

$\lambda_\boxdot$ extends λ with the modality $\boxdot$ which is concerned with *where we can use the result of evaluating a given term*. We call $\lambda_\boxdot$ a *modal language* because of its use of the modality $\boxdot$ in the type system. Although its type system addresses the safety of mobile values, $\lambda_\boxdot$ is still a language for stand-alone computations in which no communications between nodes actually take place. The modality $\boxdot$ does not originate from modal logic, but the type system of $\lambda_\boxdot$ reuses typing judgments for necessity of modal logic by Pfenning and Davies [10].

3 Modal Language $\lambda_\boxdot$

The idea behind the modality $\boxdot$ is two-fold. First, if a term M is well-typed under an empty typing context and an empty store typing, *i.e.*, $\cdot \mid \cdot \vdash M : A$, we can evaluate it at

any node. Intuitively M is valid at any node, or *globally valid*, because its evaluation depends on no existing local resources. As a special case, if a value V satisfies $\cdot \mid \cdot \vdash V : A$, it is globally valid because it does not contain references (to local resources). Second the typing judgment $\Gamma \mid \Psi \vdash M : A$ of λ is unable to express the property that the value to which term M evaluates is globally valid. Therefore we need an additional typing judgment for the type system of $\lambda_{\boxdot}$ so as to express such properties of terms.

In order to indicate that a variable stores a globally valid value, we introduce a *global typing context* Δ. Γ is now called a *local typing context*.

$$\begin{array}{lll} \text{global typing context} & \Delta & ::= \quad \cdot \mid \Delta, x \sim A \\ \text{local typing context} & \Gamma & ::= \quad \cdot \mid \Gamma, x : A \end{array}$$

A binding $x \sim A$ in Δ means that variable x holds a globally valid value of type A; hence a global typing context does not affect the mobility of a term being typechecked.

We use a typing judgment $\Delta; \Gamma \mid \Psi \vdash M : A$ to mean that under global typing context Δ, local typing context Γ, and store typing Ψ, term M evaluates to a value of type A valid at the current node; it may be viewed as a typing judgment for λ where a typing context is split into Δ and Γ. We introduce a new form of typing judgment $\Delta; \Gamma \mid \Psi \vdash M \sim A$ to mean that M evaluates to a globally valid value of type A (which is also valid at the current node). By the definition of these typing judgments, the following typing rules hold independently of the syntax of $\lambda_{\boxdot}$:

$$\frac{x \sim A \in \Delta}{\Delta; \Gamma \mid \Psi \vdash x \sim A}\ \mathsf{GVar} \qquad \frac{x \sim A \in \Delta}{\Delta; \Gamma \mid \Psi \vdash x : A}\ \mathsf{GVar'} \qquad \frac{\Delta; \cdot \mid \cdot \vdash V : A}{\Delta; \Gamma \mid \Psi \vdash V \sim A}\ \mathsf{GVal}$$

The rule GVar′ says that a globally valid variable x in $x \sim A$ is valid at the current node. The rule GVal conforms to the definition of the new typing judgment: the premise check if V is globally valid, in which case the conclusion holds because V is already a value.

The type system of $\lambda_{\boxdot}$ classifies types into three kinds: *primitive types* P, *potentially global types* G, and *local types* L. A primitive type is one for which mobility is an inherent property. For example, (), of type unit, is atomic and cannot contain references to local resources. Therefore values of type unit are always globally valid, which implies that unit is a primitive type. Formally we define primitive types as follows:

Definition 1. *P is a primitive type if and only if $\Delta; \Gamma \mid \Psi \vdash V : P$ implies $\Delta; \cdot \mid \cdot \vdash V : P$.*

By the definition of primitive types, $\Delta; \Gamma \mid \Psi \vdash M : P$ semantically implies $\Delta; \Gamma \mid \Psi \vdash M \sim P$. In order to relieve the programmer of the burden of explicitly expressing mobility for primitive types, $\lambda_{\boxdot}$ provides a separate typing rule for primitive types:

$$\frac{\Delta; \Gamma \mid \Psi \vdash M : P}{\Delta; \Gamma \mid \Psi \vdash M \sim P}\ \mathsf{Prim}{\sim}$$

In contrast to primitive types, a local type L has no globally valid values associated with it. For example, locations, of type ref A, can never be globally valid, which implies that ref A is a local type. Thus $\Delta; \Gamma \mid \Psi \vdash M \sim L$ never holds. (See Proposition 2.) A value of a potentially global type may or may not be globally valid depending on whether it contains references to local resources. For example, a function type $A \to B$ is a potentially global type.

$\lambda_{\boxdot}$ introduces two new terms $\mathsf{box}\ M$ and $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$:

$$\begin{array}{lrcl} \text{term} & M & ::= & \cdots \mid \mathsf{box}\ M \mid \mathsf{letbox}\ x = M\ \mathsf{in}\ M \\ \text{value} & V & ::= & \cdots \mid \mathsf{box}\ M \end{array}$$

$\mathsf{box}\ M$ has a modal type $\boxdot A$, and expects M to evaluate to a globally valid value. $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$ expects M to evaluate to $\mathsf{box}\ M'$; then it evaluates M' before substituting the resultant value for x in N. The β-reduction rule for the modality $\boxdot$ uses a capture-avoiding substitution $[V/x]M$ extended in a standard way.

$$\mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ M \quad \longrightarrow_\beta \quad [V/x]M$$
$$\text{evaluation context} \quad \phi \quad ::= \quad \cdots \mid \mathsf{letbox}\ x = \phi\ \mathsf{in}\ M \mid \mathsf{letbox}\ x = \mathsf{box}\ \phi\ \mathsf{in}\ M$$

$\mathsf{box}\ M$ corresponds to the introduction rule for the modality $\boxdot$. Note that in $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$, the type of M does not determine the form of the typing judgment for the whole term. That is, regardless of the type of M, there are two possibilities for where the result of evaluating N is valid: at the current node and at any node. Therefore $\boxdot$ has one introduction rule and two elimination rules:

$$\frac{\Delta; \Gamma \mid \Psi \vdash M \sim A}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}\ M : \boxdot A}\ \boxdot\mathsf{I} \qquad \frac{\Delta; \Gamma \mid \Psi \vdash M : \boxdot A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N : C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N : C}\ \boxdot\mathsf{E}$$

$$\frac{\Delta; \Gamma \mid \Psi \vdash M : \boxdot A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N \sim C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N \sim C}\ \boxdot\mathsf{E}'$$

Figure 2 summarizes the definition of $\lambda_{\boxdot}$. The operational semantics of $\lambda_{\boxdot}$ uses the same reduction judgment $M \mid \psi \longrightarrow M' \mid \psi'$ as in λ. Proposition 3 confirms that $\Delta; \Gamma \mid \Psi \vdash M \sim A$ is stronger than $\Delta; \Gamma \mid \Psi \vdash M : A$.

Proposition 2. *If no variables are bound to local types in Δ, then $\Delta; \Gamma \mid \Psi \vdash M \sim L$ is not derivable. That is, if $\Delta; \Gamma \mid \Psi \vdash M \sim A$, then A is not a local type.*

Proposition 3. *The rule* $\dfrac{\Delta; \Gamma \mid \Psi \vdash M \sim A}{\Delta; \Gamma \mid \Psi \vdash M : A}\ \mathsf{Global}$ *is admissible.*

3.1 Example

We illustrate the use of the modality $\boxdot$ by rewriting in $\lambda_{\boxdot}$ the two examples in Introduction. We assume a primitive type int for integers and an infix operator + for adding two integers. We encode a modal type $\Box A$ as $\boxdot(\mathsf{unit} \to A)$, and check the mobility of a *term* M of type A by checking the mobility of a *value* $\lambda\_ : \mathsf{unit}.\ M$ of type $\mathsf{unit} \to A$ where $\_$ denotes a fresh variable.

The first example is written in $\lambda_{\boxdot}$ as follows:

$$M_1 = (\lambda r : \mathsf{ref\ int}.\ \lambda x : \mathsf{int}.\ x + !r)\ (\mathsf{ref}\ 0)$$

We cannot use M_1 to build a term of type $\boxdot(\mathsf{int} \to \mathsf{int})$ because there is no typing derivation of $\Delta; \Gamma \mid \Psi \vdash M_1 \sim \mathsf{int} \to \mathsf{int}$:

$$\frac{\dfrac{(\textit{no typing rule applicable})}{\Delta; \Gamma \mid \Psi \vdash (\lambda r : \mathsf{ref\ int}.\ \lambda x : \mathsf{int}.\ x + !r)\ (\mathsf{ref}\ 0) \sim \mathsf{int} \to \mathsf{int}}}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}\ (\lambda r : \mathsf{ref\ int}.\ \lambda x : \mathsf{int}.\ x + !r)\ (\mathsf{ref}\ 0) : \boxdot(\mathsf{int} \to \mathsf{int})}\ \boxdot\mathsf{I}$$

$$
\begin{array}{lllll}
\text{type} & A & ::= & P \mid G \mid L \\
\text{primitive type} & P & ::= & \mathsf{unit} \\
\text{potentially global type} & G & ::= & A \to A \mid \Box A \\
\text{local type} & L & ::= & \mathsf{ref}\ A \\
\text{term} & M & ::= & \cdots \mid \mathsf{box}\ M \mid \mathsf{letbox}\ x = M\ \mathsf{in}\ M \\
\text{value} & V & ::= & \cdots \mid \mathsf{box}\ M \\
\text{global typing context} & \Delta & ::= & \cdot \mid \Delta, x \sim A \\
\text{local typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

$$
\frac{x \sim A \in \Delta}{\Delta; \Gamma \mid \Psi \vdash x \sim A}\ \mathsf{GVar} \qquad \frac{x \sim A \in \Delta}{\Delta; \Gamma \mid \Psi \vdash x : A}\ \mathsf{GVar'}
$$

$$
\frac{\Delta; \Gamma \mid \Psi \vdash M \sim A}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}\ M : \Box A}\ \Box\mathsf{I} \qquad \frac{\Delta; \Gamma \mid \Psi \vdash M : \Box A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N : C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N : C}\ \Box\mathsf{E}
$$

$$
\frac{\Delta; \Gamma \mid \Psi \vdash M : \Box A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N \sim C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N \sim C}\ \Box\mathsf{E'}
$$

$$
\frac{\Delta; \Gamma \mid \Psi \vdash M : P}{\Delta; \Gamma \mid \Psi \vdash M \sim P}\ \mathsf{Prim}\sim \qquad \frac{\Delta; \cdot \mid \cdot \vdash V : A}{\Delta; \Gamma \mid \Psi \vdash V \sim A}\ \mathsf{GVal}
$$

$$
\mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ M \quad \longrightarrow_\beta \quad [V/x]M
$$

$$
\text{evaluation context} \quad \phi \quad ::= \quad \cdots \mid \mathsf{letbox}\ x = \phi\ \mathsf{in}\ M \mid \mathsf{letbox}\ x = \mathsf{box}\ \phi\ \mathsf{in}\ M
$$

Fig. 2. Definition of the modal language $\lambda_\Box$

M_1 itself, however, is mobile because $\lambda\_ : \mathsf{unit}.\ M_1$ is mobile:

$$
\frac{\dfrac{\vdots}{\Delta; \cdot \mid \cdot \vdash \lambda\_ : \mathsf{unit}.\ (\lambda r : \mathsf{ref\ int}.\ \lambda x : \mathsf{int}.\ x + !r)\ (\mathsf{ref}\ 0) : \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})}}{\Delta; \Gamma \mid \Psi \vdash \lambda\_ : \mathsf{unit}.\ (\lambda r : \mathsf{ref\ int}.\ \lambda x : \mathsf{int}.\ x + !r)\ (\mathsf{ref}\ 0) \sim \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})}\ \mathsf{GVal}
$$

The second example is written in $\lambda_\Box$ as follows where variable r is bound to an existing reference of type $\mathsf{ref\ int}$:

$$
M_2 = \mathsf{letbox}\ v = \mathsf{box}\ !r\ \mathsf{in}\ \lambda x : \mathsf{int}.\ x + v
$$

We can use M_2 to build a term of type $\Box(\mathsf{int} \to \mathsf{int})$:

$$
\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash r : \mathsf{ref\ int}}\ \mathsf{Var}}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash !r : \mathsf{int}}\ \mathsf{Deref}}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash !r \sim \mathsf{int}}\ \mathsf{Prim}\sim}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \mathsf{box}\ !r : \Box\mathsf{int}}\ \Box\mathsf{I} \qquad \frac{\dfrac{\dfrac{\vdots}{\Delta, v \sim \mathsf{int}; x : \mathsf{int} \mid \cdot \vdash x + v : \mathsf{int}}}{\Delta, v \sim \mathsf{int}; \cdot \mid \cdot \vdash \lambda x : \mathsf{int}.\ x + v : \mathsf{int} \to \mathsf{int}}\ \to\mathsf{I}}{\Delta, v \sim \mathsf{int}; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \lambda x : \mathsf{int}.\ x + v \sim \mathsf{int} \to \mathsf{int}}\ \mathsf{GVal}}{\dfrac{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \mathsf{letbox}\ v = \mathsf{box}\ !r\ \mathsf{in}\ \lambda x : \mathsf{int}.\ x + v \sim \mathsf{int} \to \mathsf{int}}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \mathsf{box\ letbox}\ v = \mathsf{box}\ !r\ \mathsf{in}\ \lambda x : \mathsf{int}.\ x + v : \Box(\mathsf{int} \to \mathsf{int})}\ \Box\mathsf{I}}\ \Box\mathsf{E}
$$

M_2 itself, however, is not mobile because $\lambda\_ : \mathsf{unit}.\ M_2$ is not mobile:

$$
\frac{\dfrac{(\textit{impossible to typecheck because of } r)}{\Delta; \cdot \mid \cdot \vdash \lambda\_ : \mathsf{unit}.\ \mathsf{letbox}\ v = \mathsf{box}\ !r\ \mathsf{in}\ \lambda x : \mathsf{int}.\ x + v : \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})}}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \lambda\_ : \mathsf{unit}.\ \mathsf{letbox}\ v = \mathsf{box}\ !r\ \mathsf{in}\ \lambda x : \mathsf{int}.\ x + v \sim \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})}\ \mathsf{GVal}
$$

A more straightforward but less satisfactory translation of the second example uses a λ-application instead of a letbox construct:

$$M_2' = (\lambda v{:}\mathsf{int}.\, \lambda x{:}\mathsf{int}.\, x + v)\ (!r)$$

M_2' is operationally equivalent to M_2, but cannot be used in building a term of type $\boxdot(\mathsf{int} \to \mathsf{int})$:

$$\dfrac{\dfrac{(\textit{no typing rule applicable})}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash (\lambda v{:}\mathsf{int}.\, \lambda x{:}\mathsf{int}.\, x + v)\ (!r) \sim \mathsf{int} \to \mathsf{int}}}{\Delta; \Gamma, r : \mathsf{ref\ int} \mid \Psi \vdash \mathsf{box}\ (\lambda v{:}\mathsf{int}.\, \lambda x{:}\mathsf{int}.\, x + v)\ (!r) : \boxdot(\mathsf{int} \to \mathsf{int})}\ \boxdot\mathsf{I}$$

The reason why box M_2' fails to have type $\boxdot(\mathsf{int} \to \mathsf{int})$ is that the type of $\lambda v : \mathsf{int}.\, \lambda x{:}\mathsf{int}.\, x + v$, namely $\mathsf{int} \to (\mathsf{int} \to \mathsf{int})$, fails to express that a mobile value of type $\mathsf{int} \to \mathsf{int}$ is returned. In fact, the inner λ-abstraction $\lambda x : \mathsf{int}.\, x + v$ cannot be a mobile value anyway, since a binding $v \sim \mathsf{int}$ is not added to a global typing context.

We could introduce a new type $A \boxdot\!\!\to B$ for those λ-abstractions taking a value of type A and returning a mobile value of type B. The typing rules for the new connective $\boxdot\!\!\to$ are given as follows:

$$\dfrac{\Delta; \Gamma, x : A \mid \Psi \vdash M \sim B}{\Delta; \Gamma \mid \Psi \vdash \lambda x{:}A.\, M : A \boxdot\!\!\to B}\ \to\mathsf{I}_{\boxdot} \qquad \dfrac{\Delta; \Gamma \mid \Psi \vdash M : A \boxdot\!\!\to B \quad \Delta; \Gamma \mid \Psi \vdash N : A}{\Delta; \Gamma \mid \Psi \vdash M\ N \sim B}\ \to\mathsf{E}_{\boxdot}$$

Although the new connective $\boxdot\!\!\to$ allows more flexibility in programming, we decide not to include it in the definition of $\lambda_{\boxdot}$ because a simple encoding of $A \boxdot\!\!\to B$ as $A \to \boxdot B$ suffices. For example, we can eliminate the rule $\to\mathsf{I}_{\boxdot}$ by translating $\lambda x{:}A.\, M : A \boxdot\!\!\to B$ into $\lambda x{:}A.\, \mathsf{box}\ M : A \to \boxdot B$ and the rule $\to\mathsf{E}_{\boxdot}$ by translating $M\ N$ into $\mathsf{letbox}\ v = M\ N\ \mathsf{in}\ v$.

3.2 Type Safety of $\lambda_{\boxdot}$

The proof of type safety of $\lambda_{\boxdot}$ is routine except for the formulation of the substitution theorem (Theorem 4). In the second clause of the substitution theorem, $\Delta; \cdot \mid \cdot \vdash V : A$ proves that V is a globally valid value of type A, which we substitute for variable x in term M. Corollary 5 follows from the definition of primitive types as given in Definition 1.

Theorem 4 (Substitution)
If $\Delta; \Gamma \mid \Psi \vdash V : A$ *and* $\Delta; \Gamma, x : A \mid \Psi \vdash M : C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M : C$.
If $\Delta; \Gamma \mid \Psi \vdash V : A$ *and* $\Delta; \Gamma, x : A \mid \Psi \vdash M \sim C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M \sim C$.
If $\Delta; \cdot \mid \cdot \vdash V : A$ *and* $\Delta, x \sim A; \Gamma \mid \Psi \vdash M : C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M : C$.
If $\Delta; \cdot \mid \cdot \vdash V : A$ *and* $\Delta, x \sim A; \Gamma \mid \Psi \vdash M \sim C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M \sim C$.

Corollary 5
If $\Delta; \Gamma \mid \Psi \vdash V : P$ *and* $\Delta, x \sim P; \Gamma \mid \Psi \vdash M : C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M : C$.
If $\Delta; \Gamma \mid \Psi \vdash V : P$ *and* $\Delta, x \sim P; \Gamma \mid \Psi \vdash M \sim C$, *then* $\Delta; \Gamma \mid \Psi \vdash [V/x]M \sim C$.

Theorem 6 (Progress). *Suppose that term* M *satisfies* $\cdot; \cdot \mid \Psi \vdash M : A$ *or* $\cdot; \cdot \mid \Psi \vdash M \sim A$ *for some store typing* Ψ *and type* A. *Then either:*

(1) M *is a value, or*

(2) for any store ψ *such that* $\Psi \vdash \psi$ okay, *there exist some term* M' *and store* ψ' *such that* $M \mid \psi \longrightarrow M' \mid \psi'$.

Theorem 7 (Type preservation)

$$\textit{Suppose} \begin{cases} \cdot;\cdot \mid \Psi \vdash M : A \\ \Psi \vdash \psi \ \mathsf{okay} \\ M \mid \psi \longrightarrow M' \mid \psi' \end{cases}.$$

$$\textit{Then there exists a store typing } \Psi' \textit{ such that } \begin{cases} \cdot;\cdot \mid \Psi' \vdash M' : A \\ \Psi \subset \Psi' \\ \Psi' \vdash \psi' \ \mathsf{okay} \end{cases}.$$

$$\textit{Suppose} \begin{cases} \cdot;\cdot \mid \Psi \vdash M \sim A \\ \Psi \vdash \psi \ \mathsf{okay} \\ M \mid \psi \longrightarrow M' \mid \psi' \end{cases}.$$

$$\textit{Then there exists a store typing } \Psi' \textit{ such that } \begin{cases} \cdot;\cdot \mid \Psi' \vdash M' \sim A \\ \Psi \subset \Psi' \\ \Psi' \vdash \psi' \ \mathsf{okay} \end{cases}.$$

3.3 Logic for $\lambda_{\boxdot}$

The type system for modal types $\boxdot A$ is unusual in that it differentiates values (*i.e.*, terms in weak head normal form) from ordinary terms, as shown in the rule GVal. This differentiation implies that the logic corresponding to the modality $\boxdot$ via the Curry-Howard isomorphism requires a judgment that inspects not only hypotheses in a proof but also the proof structure itself (*e.g.*, inferences rules used in the proof). Thus the modality $\boxdot$ sets itself apart from other modalities and is not found in any other logic.

In the pure fragment of $\lambda_{\boxdot}$ without primitive types and local types, the modality $\boxdot$ shows similarities with modal possibility $\Diamond$ and lax modality $\bigcirc$ in [10]. Specifically a proof-theoretic analysis of $\boxdot$ gives rise to a new form of substitution $\langle M/x\rangle N$ which is defined inductively on the structure of the term being substituted (*i.e.*, M) instead of the term being substituted into (*i.e.*, N). Let us interpret a β-reduction rule as the reduction of a typing derivation in which an introduction rule is followed by a corresponding elimination rule. For example, the β-reduction rule for the connective $\rightarrow$ may be seen as the reduction of the following typing derivation in the pure λ-calculus (where we omit store typings):

$$\dfrac{\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \rightarrow B}\ \rightarrow\mathsf{I} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}A.\, M)\ N : B}\ \rightarrow\mathsf{E} \qquad \longrightarrow_{\beta} \qquad \Gamma \vdash [N/x]M : B$$

Likewise we obtain a β-reduction rule for $\boxdot$ from the reduction of a typing derivation in which the introduction rule $\boxdot\mathsf{I}$ is followed by the elimination rule $\boxdot\mathsf{E}$ or $\boxdot\mathsf{E}'$ (where we omit store typings):

$$\dfrac{\dfrac{\Delta;\Gamma \vdash M \sim A}{\Delta;\Gamma \vdash \mathsf{box}\ M : \boxdot A}\ \boxdot\mathsf{I} \qquad \Delta, x \sim A;\Gamma \vdash N : C}{\Delta;\Gamma \vdash \mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N : C}\ \boxdot\mathsf{E} \qquad \longrightarrow_{\beta} \qquad \Delta;\Gamma \vdash \langle M/x\rangle N : C$$

To see why $\langle M/x\rangle N$ is defined inductively on the structure of M, observe that the reduction of $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$ requires an analysis of M instead of N. The reason is that only a value can be substituted for x, but M may not be a value; therefore we

have to analyze M to decide how to transform the whole term so that x is eventually replaced by a value. Conceptually N should be replicated at those places within M where the evaluation of M is finished, so that M and N are evaluated exactly once and in that order. If M is already a value V, we reduce the whole term to $[V/x]N$. Thus we are led to define $\langle M/x\rangle N$ as follows:

$$\begin{array}{rcl} \langle V/x\rangle N & = & [V/x]N \\ \langle \mathsf{letbox}\ x' = M'\ \mathsf{in}\ M''/x\rangle N & = & \mathsf{letbox}\ x' = M'\ \mathsf{in}\ \langle M''/x\rangle N \end{array}$$

Note that we *cannot* define $\langle M_1\ M_2/x\rangle N$ because without primitive types and the rule $\mathsf{Prim}\sim$, there is no typing derivation of $\Delta; \Gamma \vdash M_1\ M_2 \sim A$ and thus $\mathsf{box}\ M_1\ M_2$ cannot be well-typed.

In the presence of primitive types, the β-reduction

$$\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N \longrightarrow_\beta \langle M/x\rangle N$$

is no longer valid because $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$ may typecheck while $\langle M/x\rangle N$ is undefined. For example, $M = M_1\ M_2$ of type unit satisfies $\Delta; \Gamma \vdash M \sim \mathsf{unit}$ by the rule $\mathsf{Prim}\sim$, but $\langle M_1\ M_2/x\rangle N$ is undefined. Intuitively the rule $\mathsf{Prim}\sim$ disguises an unanalyzable term of a primitive type as an analyzable term. Thus, in order to reduce $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$, the operational semantics of $\lambda_{\boxdot}$ is forced to reduce M into a value V first, instead of analyzing M to transform the whole term. Then an ordinary substitution $[V/x]N$ suffices for the reduction of $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$.

We close this section with a brief discussion of the properties of the modality $\boxdot$.

- $\boxdot A \to A$ $\qquad \lambda x{:}\boxdot A.\, \mathsf{letbox}\ y = x\ \mathsf{in}\ y$
  A mobile value is a special case of an ordinary term.
- $\boxdot A \to \boxdot\boxdot A$ $\qquad \lambda x{:}\boxdot A.\, \mathsf{letbox}\ y = x\ \mathsf{in}\ \mathsf{box}\ \mathsf{box}\ y$
  A mobile value itself is mobile.
- $\boxdot(A \to B) \to \boxdot A \twoheadrightarrow \boxdot B$
  A mobile λ-abstraction does not necessarily return a mobile value.

4 $\lambda^N_{\boxdot}$ with a Network Operational Semantics

While the type system of $\lambda_{\boxdot}$ is appropriate for understanding the role of the modality $\boxdot$, it is not expressive enough for distributed computations which may generate terms whose type is determined by *remote nodes*. For example, a future construct [11] initiates a stand-alone computation at a remote node and returns a pointer to the remote node; then the type of the pointer is determined by the term being evaluated at the remote node.

This section extends the type system of $\lambda_{\boxdot}$ so that we can typecheck such terms, and also develop a network operational semantics to model distributed computations. We refer to the resultant language as $\lambda^N_{\boxdot}$. We incorporate a communication construct for remote procedure calls into $\lambda_{\boxdot}$ so as to allow communications between nodes to actually take place. Type safety of $\lambda^N_{\boxdot}$ ensures the safety of mobile terms and mobile values.

4.1 Extended Type System and Network Operational Semantics

We represent the state of a network with a *configuration* π which records term M and store ψ associated with each node γ. A *configuration typing* Π records the type of the term being evaluated at each node. We assume that no node appears more than once in π, and consider Π as an unordered set. As a new term, γ serves as a reference to a node.

$$\begin{array}{lllll} \text{term} & M & ::= & \cdots \mid \gamma & \text{(node reference)} \\ \text{configuration} & \pi & ::= & \cdot \mid \pi, \{M \mid \psi \ @ \ \gamma\} & \\ \text{configuration typing} & \Pi & ::= & \cdot \mid \Pi, \gamma \sim A & (A \neq L) \end{array}$$

- $\{M \mid \psi \ @ \ \gamma\}$ in π means that node γ is currently evaluating term M with store ψ.
- $\gamma \sim A$ in Π means that the term at node γ evaluates to a value of type A. For the sake of simplicity, we require that every term in a network evaluates to a globally valid value.

The extended type system is formulated with a *configuration typing judgment* $\Pi \vdash \pi$ okay which means that configuration π has configuration typing Π. In order to be able to typecheck a node reference γ (which is a term), we include a configuration typing Π in each typing judgment: $\Delta; \Gamma \mid \Psi \mid \Pi \vdash M : A$, $\Delta; \Gamma \mid \Psi \mid \Pi \vdash M \sim A$, and $\Psi \mid \Pi \vdash \psi$ okay. The rules for the extended type system are derived from (and given the same name as) the previous rules by including a configuration typing in every typing judgment. We need two additional typing rules Node and Node′ for node references; the rule Conf may be regarded as the definition of the configuration typing judgment.

$$\frac{\gamma \sim A \in \Pi}{\Delta; \Gamma \mid \Psi \mid \Pi \vdash \gamma \sim A}\ \textsf{Node} \qquad \frac{\gamma \sim A \in \Pi}{\Delta; \Gamma \mid \Psi \mid \Pi \vdash \gamma : A}\ \textsf{Node}'$$

$$\frac{dom(\Pi) = dom(\pi) \qquad \begin{array}{l} \gamma \sim A \in \Pi \\ \Psi \mid \Pi \vdash \psi \text{ okay} \\ \cdot; \cdot \mid \Psi \mid \Pi \vdash M \sim A \end{array} \quad \text{for every } \{M \mid \psi \ @ \ \gamma\} \in \pi}{\Pi \vdash \pi \text{ okay}}\ \textsf{Conf}$$

The network operational semantics is formulated with a *configuration reduction judgment* $\pi \Longrightarrow \pi'$, which means that configuration π reduces (or evolves) to configuration π'. We provide two rules for the configuration reduction judgment:

$$\frac{M \mid \psi \longrightarrow M' \mid \psi'}{\pi, \{M \mid \psi \ @ \ \gamma\} \Longrightarrow \pi, \{M' \mid \psi' \ @ \ \gamma\}}\ \textit{Red}$$

$$\frac{}{\pi, \{\phi[\![\gamma']\!] \mid \psi \ @ \ \gamma\}, \{V \mid \psi' \ @ \ \gamma'\} \Longrightarrow \pi, \{\phi[\![V]\!] \mid \psi \ @ \ \gamma\}, \{V \mid \psi' \ @ \ \gamma'\}}\ \textit{Sync}$$

The rule *Red* says that stand-alone computations at individual nodes are part of a distributed computation. In the rule *Sync*, a node reference γ' suspends the stand-alone computation at node γ until it is replaced by a mobile value V through a synchronization operation with node γ'. That is, node reference γ' is *not* a value, but reduces to a mobile value (which is globally valid) only after node γ' has finished evaluating a term. Note that a configuration reduction is non-deterministic because the rule *Red* can choose an arbitrary node γ from a given configuration.

4.2 Communication Construct for Remote Procedure Calls

The network operational semantics becomes interesting only with communication constructs; without communication constructs, all nodes perform stand-alone computations independently of each other, and type safety holds trivially.

In designing communication constructs, we could begin with an existing modal logic, such as S4 and S5, and adhere to a spatial interpretation of the modalities in it. Then there arise a few logically motivated primitive operations, with which we can implement various communication constructs. This approach is appealing because of the strong logical foundation underlying communication constructs as well as the pleasant correspondence between modal logic and distributed computations.

A strict adherence to a spatial interpretation of modal logic, however, sacrifices flexibility in programming. For example, all previous work on modal languages for distributed computations [6,7,8] builds on the idea of using modal types $\Box A$ for mobile *terms* that may be evaluated at any node in the network, which is a typical spatial interpretation of the modality $\Box$. As modal types $\Box A$ do not ensure the safety of mobile values and safe communications are thus restricted to mobile terms, it is difficult to give a faithful implementation of those communication constructs (such as remote procedure calls, future constructs, and communication channels) that expect or return mobile values.

Since it is not concerned with "how" to transmit mobile values between nodes, the modality $\boxdot$ itself does not specify a principle for the design of communication constructs in $\lambda^N_{\boxdot}$. Instead we have to design each communication construct individually by exploiting node references γ in conjunction with the rule *Sync*. We do not believe that our approach is *ad hoc*, since not every communication construct can be given a logical interpretation anyway (*e.g.*, communication channels). In fact, even λ_{rpc} [7] and *Lambda 5* [8], both of which are based on a spatial interpretation of modal logic S5, introduce primitive operations which are logically "motivated," but do not actually have their counterparts in S5.

As an illustration, we develop a communication construct for remote procedure calls. We use the natural deduction style by giving introduction and elimination rules for it. (Thus our communication construct is also logically motivated to a certain extent.) We can use the same idea to develop similar communication constructs, such as future constructs, that transmit both mobile terms and mobile values.

A remote procedure call transmits a mobile term M to a remote node γ to initiate a stand-alone computation, and then waits for the result of evaluating M. In order to ensure the safety of the remote procedure call, M needs to satisfy the following two conditions:

- M itself is globally valid so that the evaluation of M at node γ is safe.
- M evaluates to a globally valid value so that the result of the remote procedure call is valid.

We can test if M satisfies the two conditions by typechecking $\mathsf{box}\ \mathsf{box}\ M$:

- Typechecking the outer box construct tests if $\mathsf{box}\ M$ evaluates to a globally valid value, in which case M is also globally valid because $\mathsf{box}\ M$ is already a value.
- Typechecking the inner box construct tests if M evaluates to a globally valid value.

Thus, under typing contexts Δ and Γ and store typing Ψ, we have to prove $\Delta; \cdot \mid \cdot \vdash M \sim A$:

$$\dfrac{\dfrac{\dfrac{\Delta; \cdot \mid \cdot \vdash M \sim A}{\Delta; \cdot \mid \cdot \vdash \mathsf{box}\ M : \boxdot A}\ \boxdot\mathsf{I}}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}\ M \sim \boxdot A}\ \mathsf{GVal}}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}\ \mathsf{box}\ M : \boxdot\boxdot A}\ \boxdot\mathsf{I}$$

We do not, however, use a term $\mathsf{box}\ \mathsf{box}\ M$ of type $\boxdot\boxdot A$ for a remote procedure call because there is no way to tell whether such a term is intended for a remote procedure call or just for creating a globally valid value. Instead we introduce a new modality $\boxdot^2$ specifically designed for remote procedure calls. As far as the type system is concerned, we may think of $\boxdot^2 A$ as an abbreviation of $\boxdot\boxdot A$. We use $\mathsf{box}^2\ M$ and $\mathsf{letbox}^2\ x = M\ \mathsf{in}\ N$ for the introduction and elimination rules for $\boxdot^2$, respectively:

potentially global type	G	::=	$\cdots \mid \boxdot^2 A$
term	M	::=	$\cdots \mid \mathsf{box}^2\ M \mid \mathsf{letbox}^2\ x = M\ \mathsf{in}\ M$
value	V	::=	$\cdots \mid \mathsf{box}^2\ M$

$$\dfrac{\Delta; \cdot \mid \cdot \vdash M \sim A}{\Delta; \Gamma \mid \Psi \vdash \mathsf{box}^2\ M : \boxdot^2 A}\ \boxdot^2\mathsf{I} \qquad \dfrac{\Delta; \Gamma \mid \Psi \vdash M : \boxdot^2 A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N : C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}^2\ x = M\ \mathsf{in}\ N : C}\ \boxdot^2\mathsf{E}$$

$$\dfrac{\Delta; \Gamma \mid \Psi \vdash M : \boxdot^2 A \quad \Delta, x \sim A; \Gamma \mid \Psi \vdash N \sim C}{\Delta; \Gamma \mid \Psi \vdash \mathsf{letbox}^2\ x = M\ \mathsf{in}\ N \sim C}\ \boxdot^2\mathsf{E}'$$

In the rules $\boxdot^2\mathsf{E}$ and $\boxdot^2\mathsf{E}'$, it helps to think of $\mathsf{letbox}^2\ x = M\ \mathsf{in}\ N$ as $\mathsf{letbox}\ x' = M\ \mathsf{in}\ \mathsf{letbox}\ x = x'\ \mathsf{in}\ N$ where x' is a fresh variable.

As for the operational semantics, $\boxdot^2 A$ diverges from $\boxdot\boxdot A$. $\mathsf{letbox}^2\ x = M\ \mathsf{in}\ N$ at node γ expects M to evaluate to $\mathsf{box}^2\ M'$; then it makes a remote procedure call by starting an evaluation of M' at a fresh node γ' and replacing M' by a node reference γ'. When the remote procedure call returns a (globally valid) mobile value V, we replace node reference γ' by V, and then reduce $\mathsf{letbox}^2\ x = \mathsf{box}^2\ V\ \mathsf{in}\ N$ to $[V/x]N$.

$$\mathsf{letbox}^2\ x = \mathsf{box}^2\ V\ \mathsf{in}\ M \quad \longrightarrow_\beta \quad [V/x]M$$

$$\text{evaluation context} \quad \phi ::= \cdots \mid \mathsf{letbox}^2\ x = \phi\ \mathsf{in}\ M \mid \mathsf{letbox}^2\ x = \mathsf{box}^2\ [\,]\ \mathsf{in}\ M$$

$$\dfrac{M \neq \gamma' \quad \text{fresh node reference } \gamma'}{\pi, \left\{\phi[\![\mathsf{letbox}^2\ x = \mathsf{box}^2\ M\ \mathsf{in}\ N]\!] \mid \psi\ @\ \gamma\right\} \Longrightarrow \pi, \left\{\phi[\![\mathsf{letbox}^2\ x = \mathsf{box}^2\ \gamma'\ \mathsf{in}\ N]\!] \mid \psi\ @\ \gamma\right\}, \{M \mid \cdot\ @\ \gamma'\}}\ RPC$$

In the extended definition of evaluation contexts, an important restriction is that the hole in $\mathsf{letbox}^2\ x = \mathsf{box}^2\ [\,]\ \mathsf{in}\ M$ can be filled only with a node reference. For example, $(\mathsf{letbox}^2\ x = \mathsf{box}^2\ [\,]\ \mathsf{in}\ M)[\![\gamma]\!]$ is allowed, but $(\mathsf{letbox}^2\ x = \mathsf{box}^2\ [\,]\ \mathsf{in}\ M)[\![N]\!]$ is not allowed. Without this restriction, $\mathsf{letbox}^2\ x = \mathsf{box}^2\ N\ \mathsf{in}\ M$ reduces to $\mathsf{letbox}^2\ x = \mathsf{box}^2\ N'\ \mathsf{in}\ M$ without making a remote procedure call, if N reduces to N'. Note also that $\mathsf{letbox}^2\ x = \mathsf{box}^2\ \gamma'\ \mathsf{in}\ M$ does not reduce to $[\gamma'/x]M$ because node reference γ' is not a value.

Independently of the modality $\boxdot^2$, the modality $\boxdot$ is still useful for creating arguments to remote procedure calls. That is, we use $\boxdot$ to compose mobile terms for remote

procedure calls, and $\boxdot^2$ to transmit them to remote nodes. For example, the following term makes a remote procedure call to add two integers n_1 and n_2, both of which are bound to variables x_1 and x_2 via letbox constructs:

$$\mathsf{letbox}\ x_1 = \mathsf{box}\ n_1\ \mathsf{in}\ \mathsf{letbox}\ x_2 = \mathsf{box}\ n_2\ \mathsf{in}\ \mathsf{letbox}^2\ v = \mathsf{box}^2\ x_1 + x_2\ \mathsf{in}\ v$$

4.3 Type Safety of $\lambda^N_{\boxdot}$

Type safety of $\lambda^N_{\boxdot}$ consists of *configuration progress* (Theorem 9) and *configuration typing preservation* (Theorem 11). Proofs of Theorems 9 and 11 use type safety for stand-alone computations in $\lambda^N_{\boxdot}$ (Theorems 8 and 10).

Theorem 8 (Progress). *Suppose that term M satisfies $\cdot;\cdot \mid \Psi \mid \Pi \vdash M : A$ or $\cdot;\cdot \mid \Psi \mid \Pi \vdash M \sim A$ for some store typing Ψ, configuration typing Π, and type A. Then one of the following holds:*

(1) M is a value,

(2) $M = \phi[\![\gamma]\!]$,

(3) $M = \phi[\![\mathsf{letbox}^2\ x = \mathsf{box}^2\ M'\ \mathsf{in}\ N]\!]$,

(4) for any store ψ such that $\Psi \mid \Pi \vdash \psi$ okay, there exist some term M' and store ψ' such that $M \mid \psi \longrightarrow M' \mid \psi'$.

Theorem 9 (Configuration progress). *Suppose $\Pi \vdash \pi$ okay. Then either:*

(1) π consists only of

$\{V \mid \psi\ @\ \gamma\}$,

$\{\phi[\![\gamma']\!] \mid \psi\ @\ \gamma\}$,

$\left\{\phi[\![\mathsf{letbox}^2\ x = \mathsf{box}^2\ \gamma'\ \mathsf{in}\ N]\!] \mid \psi\ @\ \gamma\right\}$, *or*

(2) there exists π' such that $\pi \Longrightarrow \pi'$.

Theorem 10 (Type preservation)

$$\textit{Suppose}\begin{cases}\cdot;\cdot \mid \Psi \mid \Pi \vdash M : A\\ \Psi \mid \Pi \vdash \psi\ \mathsf{okay}\\ M \mid \psi \longrightarrow M' \mid \psi'\end{cases}.$$

$$\textit{Then there exists a store typing } \Psi' \textit{ such that } \begin{cases}\cdot;\cdot \mid \Psi' \mid \Pi \vdash M' : A\\ \Psi \subset \Psi'\\ \Psi' \mid \Pi \vdash \psi'\ \mathsf{okay}\end{cases}.$$

$$\textit{Suppose}\begin{cases}\cdot;\cdot \mid \Psi \mid \Pi \vdash M \sim A\\ \Psi \mid \Pi \vdash \psi\ \mathsf{okay}\\ M \mid \psi \longrightarrow M' \mid \psi'\end{cases}.$$

$$\textit{Then there exists a store typing } \Psi' \textit{ such that } \begin{cases}\cdot;\cdot \mid \Psi' \mid \Pi \vdash M' \sim A\\ \Psi \subset \Psi'\\ \Psi' \mid \Pi \vdash \psi'\ \mathsf{okay}\end{cases}.$$

Theorem 11 (Configuration typing preservation)

$$\textit{Suppose}\begin{cases}\Pi \vdash \pi\ \mathsf{okay}\\ \pi \Longrightarrow \pi'\end{cases}.$$

$$\textit{Then there exists a configuration typing } \Pi' \textit{ such that } \begin{cases}\Pi \subset \Pi'\\ \Pi' \vdash \pi'\ \mathsf{okay}\end{cases}.$$

Type safety of $\lambda^N_{\boxdot}$ implies that mobile terms and mobile values are both safe to use: well-typed terms never go wrong even in the presence of mobile terms and mobile values.

5 Related Work

Borghuis and Feijs [5] present a typed λ-calculus *MTSN* (Modal Type System for Networks) which assumes stationary services (*i.e.*, stationary code) and mobile data. An indexed modal type $\boxdot^{\omega}(A \to B)$ represents services transforming data of type A into data of type B at node ω. MTSN is a task description language rather than a programming language, since services are all "black boxes" whose inner workings are unknown. For example, terms of type *tex* $\to$ *dvi* all describe procedures to convert *tex* files to *dvi* files. Thus reduction on terms is tantamount to simplifying procedures to achieve a certain task.

Jia and Walker [7] present a modal language λ_{rpc} which is based on hybrid logic [12] as every typing judgment explicitly specifies the current node where typechecking takes place. The modalities $\Box$ and $\Diamond$ are used for mobile terms that can be evaluated at any node and at a certain node, respectively.

Murphy *et al.* [8] present a modal language *Lambda 5* which addresses both code mobility and resource locality. It is based on modal logic S5 where all judgments are relativized to nodes, as in Simpson [13]. A value of type $\Box A$ contains a mobile term that can be evaluated at any node, and a value of type $\Diamond A$ contains a *label*, a reference to a local resource. A label may appear at remote nodes, but the type system guarantees that it is dereferenced only at the node where it is valid.

λ_{rpc} and *Lambda 5* are fundamentally different from $\lambda^N_{\boxdot}$ in their use of modal types $\Box A$ for remote procedure calls. In both languages, a remote procedure call, by the pull construct in λ_{rpc} and by the fetch construct in *Lambda 5*, is given a specific node where the evaluation is to occur, and therefore *does not expect a term contained in a value of type* $\Box A$. Instead it expects just a term of type $\Box A$, which itself may not be mobile but eventually produces a mobile term valid at any node including the caller node. The resultant mobile term is delivered to (*i.e.*, pulled or fetched by) the caller node, which needs to further evaluate it to obtain a value. As such, both languages do not address the issue of value mobility. In contrast, a remote procedure call in $\lambda^N_{\boxdot}$ transmits a term *contained* in a value of type $\boxdot^2 A$ and relies on the modality $\boxdot^2$ to directly return a mobile value.

Moody [6] presents a system which is based on modal logic S4. The modality $\Box$ is used for mobile terms that can be evaluated at any node, and the modality $\Diamond$ is used for terms located at some node. As in λ_{rpc} and *Lambda 5*, remote procedure calls use modal types $\Box A$ to transmit mobile terms to unknown remote nodes. Moody's system uses the elimination rules for the modalities $\Box$ and $\Diamond$ to send mobile terms to remote nodes, and does not provide a separate construct for remote procedure calls.

Liblit and Aiken [14] give a type-theoretic analysis of pointers in distributed computations. Their type systems distinguish between global pointers (for global address space) and local pointers (for local address space), and deal with safety and performance issues with global pointer dereferencing. While the use of type qualifiers gives a powerful type inference system for minimizing the number of global pointers, their language

focuses on distributed data rather than mobile code, and it is not obvious whether their type systems can be extended to include mobile code.

6 Conclusion and Future Work

We present a modal language $\lambda^N_{\boxdot}$ for distributed computations which ensure the safety of both mobile terms and mobile values with a single modality $\boxdot$. The modality $\boxdot$ is more expressive than the necessity modality $\Box$ from modal logic, and enables us to achieve a more faithful implementation of common communication constructs. $\boxdot$ is, however, useful in $\lambda^N_{\boxdot}$ only because the unit of communication includes values. That is, if the unit of communication was just terms and did not include values, $\Box$ would be enough and $\boxdot$ would be unnecessary.

A drawback of $\lambda^N_{\boxdot}$ is that references to local resources cannot be transmitted to remote nodes. As an example, consider a location l of type $\mathsf{ref}\ A$ at node γ. Node γ wishes to share l among all its child nodes, *e.g.*, those nodes created by remote procedure calls. No child node, however, even knows the existence of l because references to local resources cannot escape their host nodes.

To overcome this drawback, we are currently investigating another modality $\diamond\!\!\!\!\diamond$ which is similar to the modality $\diamond$ of λ_{rpc} [7], *but focuses on values rather than terms*. The idea is that term M in $\mathsf{dia}\ M$ of type $\diamond\!\!\!\!\diamond A$ evaluates to a value valid at a certain node that is unknown to the type system but known to the runtime system. The modality $\diamond\!\!\!\!\diamond$ makes it possible for mobile terms to contain references to remote resources, thereby allowing more flexibility in programming for distributed computations.

References

1. Cardelli, L.: A language with distributed scope. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1995) 286–297
2. Knabe, F.C.: Language Support for Mobile Agents. PhD thesis, Department of Computer Science, Carnegie Mellon University (1995)
3. Mascolo, C., Picco, G.P., Roman, G.C.: A fine-grained model for code mobility. In: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Springer-Verlag (1999) 39–56
4. Sekiguchi, T., Yonezawa, A.: A calculus with code mobility. In: FMOODS '97: Proceeding of the IFIP TC6 WG6.1 International Workshop on Formal Methods for Open Object-based Distributed Systems, Chapman & Hall, Ltd. (1997) 21–36
5. Borghuis, T., Feijs, L.: A constructive logic for services and information flow in computer networks. The Computer Journal **43**(4) (2000) 275–289
6. Moody, J.: Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University (2003)
7. Jia, L., Walker, D.: Modal proofs as distributed programs (extended abstract). In Schmidt, D., ed.: Proceedings of the European Symposium on Programming, LNCS 2986, Springer (2004) 219–233

8. Murphy, VII, T., Crary, K., Harper, R., Pfenning, F.: A symmetric modal lambda calculus for distributed computing. In: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004), IEEE Press (2004)
9. Park, S.: A modal language for the safety of mobile values. Technical Report POSTECH-CSE-06-001, Department of Computer Science and Engineering, Pohang University of Science and Technology (2006)
10. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science **11**(4) (2001) 511–540
11. Halstead, Jr., R.H.: Multilisp: a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems **7**(4) (1985) 501–538
12. Braüner, T.: Natural deduction for hybrid logic. Journal of Logic and Computation **14**(3) (2004) 329–353
13. Simpson, A.K.: The Proof Theory and Semantics of Intuitionistic Modal Logic. PhD thesis, Department of Philosophy, University of Edinburgh (1994)
14. Liblit, B., Aiken, A.: Type systems for distributed data structures. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2000) 199–213