Semantics for Communication Primitives in a Polymorphic Language*

Atsushi Ohori
Oki Electric Industry
Kansai Laboratory
Crystal Tower, 1-2-27 Shiromi
Chuo-ku, Osaka 540, JAPAN

email : ohori@kansai.oki.co.jp

Kazuhiko Kato
University of Tokyo
Department of Information Science
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, JAPAN

email : kato@is.s.u-tokyo.ac.jp

Abstract

We propose a method to extend an ML-style polymorphic language with transparent communication primitives, and give their precise operational semantics. These primitives allow *any* polymorphic programs definable in ML to be used remotely in a manner completely transparent to the programmer. Furthermore, communicating programs may be based on different architecture and use different data representations.

We define a polymorphic functional calculus with transparent communication primitives, which we call **dML**, as an extension of Damas and Milner's proof system for ML. We then develop an algorithm to translate **dML** to a "core" language containing only low-level communication primitives that are readily implementable in most of distributed environments. To establish the *type safety* of communicating programs, we define an operational semantics of the core language and prove that the polymorphic type system of **dML** is sound with respect to the operational semantics of the translated terms of the core language.

1 Introduction

Constructing a distributed application is a difficult task. Writing a communicating program requires complicated code to interact with the system's low-level communication primitives. Moreover, the data structures available for communication are rather limited and usually do not match those required for the application. As a result, the programmer must encode necessary data structures in primitive form. This situation is particularly unfortunate when the programmer cannot make full use of statically checked polymorphic type systems, most notably that of ML [MTH90]. If transparent communication primitives were uniformly integrated in a type system of a polymorphic programming language, then development of a distributed application would be significantly simpler and more reliable.

There are a number of models for distributed computation [BST89]. For example, one may share a primary memory and other may be coupled with a special programming language. Accordingly, there are various approaches to designing a distributed programming language. Our focus here is to develop a method to extend a general-purpose programming language with transparent communication primitives so that it can be used in a wide range of distributed environments. In particular, we would like to use a distributed language in a *heterogeneous* distributed environment, where processors with different architecture and different data representations are connected by a communication network. To achieve such generality, we minimize our assumption on underlying operating system support for distributed communication and assume that the data that can be physically exchanged between processors are restricted to atomic data. Typically, they are data of base types such as integers or boolean values. Those data can be easily encoded in a standard format and be safely transmitted through communication networks. Our goal is then to develop high-level transparent communication primitives based on this very restrictive assumption.

One useful concept for designing such a language is *remote procedure call* [BN84], which allows the programmer to use procedures (or functions) defined in a different program session residing in a different site. The advantages of this mechanism have been widely recognized and several distributed programming systems have been implemented based on this mechanism (see [BST89] for

*Appeared in *Proceedings of 20th ACM POPL Symposium, 1993.*

a survey.) Unfortunately, however, this mechanism has not yet been well integrated in type systems of programming languages, and the types of functions that can be called remotely are still very limited. Despite some works [JR86, LBG$^+$87, HS87] to extend this technique, there seems to be no systematic method that achieves transparent call of higher-order functions. Cooper and Krumvieda [CK92] proposed distributed primitives for Standard ML based on a form of remote procedure call mechanism developed for C. They suggested certain degree of remote access of polymorphic programs by giving polymorphic types to their primitives. However, they only gave an explanation of these primitives for the case of a base type, and did not provide any account for the cases of higher-order types or polymorphic types. To the authors' knowledge, there is no successful proposal that achieves transparent call of polymorphic functions in a heterogeneous distributed environment. Furthermore, there seems to be no formal attempt to establish the type safety of a distributed language.

The goals of this paper are: (1) to extend an ML style polymorphic language with transparent communication primitives, (2) to give their precise operational semantics, and (3) to prove that our extension preserves the soundness of the polymorphic type system of ML, which guarantees the type safety of communicating programs. While we focus on one language, namely ML, our method does not require communicating programs to be written in the same language. Since our method does not assume any data representation specific to ML, different languages, once extended by our method, can exchange data to the extent that the types of the data are representable in each of the languages involved. The rationale of presenting our method through ML is that its type system provides various desirable features including higher-order functions and polymorphism, and can serve as a model of advanced type systems. Monomorphic portion of our method, for example, can be used to extend other statically typed monomorphic languages.

We define a polymorphic language with transparent communication primitives, which we call **dML**, as an extension of Damas and Milner's [DM82] proof system for ML. In addition to ML term constructors, it contains a declaration constructor:

`export` e `as` x

which makes the value e accessible from remote sites through the name x, and a term constructor:

`import` x:σ `in` e `end`

which binds x in e to a remote datum of type σ (made available by an `export` declaration). This syntactically simple addition of `export` and `import` statements to ML has a significant practical implication that it allows the programmer to use remote data of any polymorphic type σ in exactly the same way as one use polymorphic programs by ML's let binding. Developing a theoretically sound method to implement these statements constitutes a major technical challenge which we claim has never been successfully met. For example, how do we export a polymorphic higher-order function to a remote site that has different architecture? Our main technical contribution is to establish one such method by developing an algorithm to translate **dML** to a "core" language that can be implemented by using a conventional mechanism of remote procedure call as described in [BN84]. To verify the correctness of our translation and establish the type safety of **dML**, we define an operational semantics of the core language and prove that **dML** polymorphic type system is sound with respect to the operational semantics of the translated term of the core language.

In addition to transparent communication, there are two important issues in distributed computation. They are *concurrency* and *recovery from failure*. For the first point, it is natural to couple communication with asynchronous computation to exploit concurrency in distributed programming. However, since the primary purpose of this paper is to give a systematic method for transparent use of polymorphic programs between different systems, we will not deal with the issue of concurrency and restrict our consideration to synchronous communication in a simple client-server style model. We believe that a mechanism of concurrency is mostly orthogonal to a mechanism of transparent communication, and that our method can be combined with an appropriate mechanism of concurrency control such as [Rep91, BMT92] to design a concurrent distributed programming language. For the second issue, although we claim that the operational semantics of the core language presented in this paper can be implemented in most distributed environments, an actual implementation requires careful considerations of recovery from failure. Here we ignore this issue and assume that interprocess communication is atomic and always succeeds. We believe that existing approaches such as [CK92] can be used to implement our core language.

The rest of the paper is organized as follows. Section 2 gives the definition of **dML**. Section 3 explains our strategy. Section 4 defines the core language. Section 5 gives the translation algorithm. Section 6 defines an operational semantics of the core language, and proves the soundness of the type system of **dML** with respect to the operational semantics of the translated terms. Our method can be extended to deal with abstract data types. Section 7 discuss this extension. Section 8 describes an application of our method to *shared* persistent programming. By combining our method with dynamic types [ACPP91], it is possible to develop a per-

sistent system that can be shared in a heterogeneous environment. Section 9 concludes the paper by discussing topics of further investigations.

2 Definition of the Language dML

As in [DM82], the set of types is divided into the set of *monotypes* (ranged over by τ) and the set of *polytypes* (ranged over by σ) given by the following abstract syntax:

$$\tau ::= t \mid b \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall t.\sigma$$

where t stands for type variables and b for base types. The set of raw terms is given by the syntax:

$$e ::= c^b \mid x \mid \lambda x.e \mid (e\ e)$$
$$\mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e\ \texttt{end}$$
$$\mid \texttt{import}\ x{:}\sigma\ \texttt{in}\ e\ \texttt{end}$$

c^b denotes constants of base type b. A top-level statement of the language is either a closed term having a closed type, or an export declaration of the form:

$$\texttt{export}\ e\ \texttt{as}\ x{:}\sigma$$

which allows the program e to be used remotely by the name x of type σ. If the type specification "$:\sigma$" is omitted, then the principal type of e is assumed. The identifier x in $\texttt{import}\ x{:}\sigma\ \texttt{in}\ M\ \texttt{end}$ plays double roles. It is a variable bound in M to a remote program; it is also an external name used to establish the binding of x to a remote program made available by the `export` declaration containing the same name x. For the simplicity, we assume that an external name is unique in the entire network.

The type system of this calculus is given by specifying a set of rules to derive a *typing* of the form:

$$\mathcal{A} \rhd e : \sigma$$

where $\mathcal{A}$ is a type assignment, which is a function from a finite set of variables to polytypes. Given a function $\mathcal{A}$, we write $\mathcal{A}\{x \mapsto \sigma\}$ for the function $\mathcal{A}'$ such that $domain(\mathcal{A}') = domain(\mathcal{A}) \cup \{x\}$, and for any $y \in domain(\mathcal{A}')$ if $y = x$ then $\mathcal{A}'(y) = \sigma$, otherwise $\mathcal{A}'(y) = \mathcal{A}(y)$. The set of typing rules is given in Figure 1. In (tapp), the notation $\sigma[\tau/t]$ denotes the type obtained from σ by substituting each occurrence of t with τ (with the necessary bound type variable renaming.) In (export), an `export` statement is not a term but a declaration, whose effect is a modification of a network wide name binding.

Note that the rule (import) allows transparent use of remote data: in the body e of $\texttt{import}\ x{:}\sigma\ \texttt{in}\ e\ \texttt{end}$

(const) $\mathcal{A} \rhd c^b : b$

(var) $\mathcal{A} \rhd x : \sigma$ if $x \in domain(\mathcal{A}), \mathcal{A}(x) = \sigma$

(abs) $$\frac{\mathcal{A}\{x \mapsto \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.e : \tau_1 \to \tau_2}$$

(app) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

(tabs) $$\frac{\mathcal{A} \rhd e : \sigma}{\mathcal{A} \rhd e : \forall t.\sigma}$$ if t does not appear in $\mathcal{A}$

(tapp) $$\frac{\mathcal{A} \rhd e : \forall t.\sigma}{\mathcal{A} \rhd e : \sigma[\tau/t]}$$

(let) $$\frac{\mathcal{A} \rhd e_1 : \sigma \qquad \mathcal{A}\{x \mapsto \sigma\} \rhd e_2 : \tau}{\mathcal{A} \rhd \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

(import) $$\frac{\mathcal{A}\{x \mapsto \sigma\} \rhd e : \tau}{\mathcal{A} \rhd \texttt{import}\ x{:}\sigma\ \texttt{in}\ e\ \texttt{end} : \tau}$$

(export) $\texttt{export}\ e\ \texttt{as}\ x{:}\sigma$ is well typed if $\emptyset \rhd e : \sigma$ is derivable

Figure 1: **dML** Type System

the identifier x can be used as an ordinary expression of type σ without any restriction. As a simple example, one site may define a function and make it available to remote sites by an export declaration, as in:

```
export λf.λx.(f (f x)) as TWICE
```

A program in a different site can use this function, as in:

```
import TWICE : ∀t.(t → t) → t → t in
   let addone = λx.x + 1 in
      ((TWICE addone) 3)
   end
end
```

2.1 Type Inference and Explicitly Typed Terms

The type system of **dML** is essentially the same as that of the Damas-Milner system, and Milner's [Mil78] algorithm $\mathcal{W}$ can be used to infer a principal type of any typable raw term. The following property was shown in [DM82].

Theorem 1 (Damas-Milner) *There is an algorithm $\mathcal{W}$ which, given $\mathcal{A}$ and e, returns either a pair (θ, τ) of a substitution θ and a monotype τ, or failure such that*

(1) if it returns (θ, τ) then $\theta(\mathcal{A}) \rhd e : \tau$ is derivable and for any $\mathcal{A}'$ and σ' if $\mathcal{A}' \rhd e : \sigma'$ then there is some substitution θ' such that $\mathcal{A}'(x) = \theta'(\theta(\mathcal{A}(x)))$ for all $x \in domain(\mathcal{A})$, and σ is a generic instance of $\theta'(\sigma')$ where σ' is the closure of τ with respect to the type assignment $\theta(\mathcal{A})$, and (2) if $\mathcal{W}$ returns failure then there is no θ' and σ' such that $\theta'(\mathcal{A}) \rhd e : \sigma'$. ∎

As shown by Mitchell and Harper [MH88], the algorithm can be extended to infer an *explicitly typed term* corresponding to a typing derivation constructed during a type inference process. This extension is useful to define a translation algorithm in Section 5. The explicitly typed **dML** contains the following set of explicitly typed terms:

$$\begin{aligned} M ::= {} & \texttt{(}c^b\texttt{:}b\texttt{)} \mid x \mid \lambda x\texttt{:}\tau\texttt{.}M \mid \texttt{(}M\ M\texttt{)} \\ & \mid \lambda t\texttt{.}M \mid \texttt{(}M\ \tau\texttt{)} \\ & \mid \texttt{let } x\texttt{:}\sigma \texttt{ = } M \texttt{ in } M \texttt{ end} \\ & \mid \texttt{import } x\texttt{:}\sigma \texttt{ in } M \texttt{ end} \end{aligned}$$

and export declarations of the form:

$$\texttt{export } M \texttt{ as } x\texttt{:}\sigma$$

$\lambda t\texttt{.}M$ is type abstraction and $\texttt{(}M\ \tau\texttt{)}$ is type application. The typing rules for the explicitly typed **dML** are obtained form those of **dML** by replacing the term of the conclusion in the rules (abs), (tabs), (tapp), and (let) with the corresponding explicitly typed term. For example, rule (tabs) becomes:

$$\text{(tabs)}\ \frac{\mathcal{A} \rhd M : \sigma}{\mathcal{A} \rhd \lambda t\texttt{.}M : \forall t.\sigma} \quad \text{if } t \text{ not appear in } \mathcal{A}$$

The type erasure $er(M)$ of an explicitly typed term M is the raw term obtained from M defined by the following rules:

$$\begin{aligned} er(\texttt{(}c^b\texttt{:}b\texttt{)}) &= c^b \\ er(x) &= x \\ er(\lambda x : \tau.M) &= \lambda x.er(M) \\ er(\texttt{(}M_1\ M_2\texttt{)}) &= \texttt{(}er(M_1)\ er(M_2)\texttt{)} \\ er(\lambda t.M) &= er(M) \\ er(\texttt{(}M\ \tau\texttt{)}) &= er(M) \\ er(\texttt{let } x\texttt{:}\sigma \texttt{ = } M_1 \texttt{ in } M_2 \texttt{ end}) & \\ &= \texttt{let } x \texttt{ = } er(M_1) \texttt{ in } er(M_2) \texttt{ end} \\ er(\texttt{import } x\texttt{:}\sigma \texttt{ in } M \texttt{ end}) & \\ &= \texttt{import } x\texttt{:}\sigma \texttt{ in } er(M) \texttt{ end} \\ er(\texttt{export } M \texttt{ as } x\texttt{:}\sigma) & \\ &= \texttt{export } er(M) \texttt{ as } x\texttt{:}\sigma \end{aligned}$$

Theorem 2 (Mitchell-Harper) *For any raw term e, if $\mathcal{A} \rhd e : \sigma$ then there is an explicitly typed term M such that $er(M) = e$ and $\mathcal{A} \rhd M : \sigma$ is derivable in the explicit calculus. Furthermore, M can be constructed efficiently from a derivation of $\mathcal{A} \rhd e : \sigma$.* ∎

Combining these two theorems, we can extend Milner's algorithm $\mathcal{W}$ so that it infers from a given raw term its principal typing and the corresponding explicitly typed term. Later we make use of this to construct a translation algorithm from **dML** to the core language.

3 Strategy for Defining a Semantics of dML

We aim to develop an algorithm to translate **dML** to the core language. One role of the core language is to define a precise operational semantics of **dML**. The second is to provide an effective algorithm to implement **dML** for a wide range of distributed environments. To achieve the second goal, we need to make assumptions about communication mechanisms available in most distributed environments.

We require data that can be transmitted between sites to have a *communication type*, which is either a base type in canonical representation (denoted by $\overline{b}$) or an *identifier* type (denoted by $id(\sigma)$). A value of type $\overline{b}$ is a value of type b encoded in common data format supported by the underlying communication protocol. A value of $id(\sigma)$ is an identifier of a value of type σ. Identifiers can be any fixed length data and they are treated as atomic data during communication. They will never be interpreted in a remote system. An identifier sent from a remote system is used only as an argument to the primitive of remote function application described below. For communication, we assume that the underlying system provides the following mechanisms:

1. a network-wide *name server* to bind a name to a remote datum of a communication type, and

2. a mechanism to apply a remote function (passed as an identifier from a remote site) to a value of a communication type, and to receive the result which must also have a communication type.

With the restriction on types of data, these primitives are implementable in most distributed systems. In particular, the mechanism of applying remote functions can be implemented by the standard remote procedure call mechanism [BN84].

Under these assumptions, we develop an algorithm to translate `export` and `import` statements into terms of the core language. The idea is to define for each type σ, the communication type $\mathcal{C}(\sigma)$ representing σ, and then define an "encoder" and "decoder" pair for each type σ that encodes values of type σ into values of type $\mathcal{C}(\sigma)$ and decodes values of type $remote(\mathcal{C}(\sigma))$ back to values of type σ. Type $remote(\mathcal{C}(\sigma))$ denotes values of type $\mathcal{C}(\sigma)$ created in and passed from a (possibly) different site.

For a base type b, we assume that primitive functions $\texttt{encoder}^b$ and $\texttt{decoder}^b$ to convert data formats are supplied by the system. For a function f of type $\tau_1 \to \tau_2$, the encoding is done at the exporting site by creating an identifier of a new function F of type $remote(\mathcal{C}(\tau_1)) \to \mathcal{C}(\tau_2)$ from f. Function F first decodes its argument of type $remote(\mathcal{C}(\tau_1))$ to get a value of type τ_1, then applies the given function f to it, finally encodes the result of the application in type $\mathcal{C}(\tau_2)$. At the importing site, the corresponding decoding is done by creating a new function G of type $\tau_1 \to \tau_2$ from the identifier of F passed from the exporting site. The function G first encodes the given argument of type τ_1 into type $\mathcal{C}(\tau_1)$, then invokes the remote function call mechanism to apply F to the encoded argument and receives the result of type $\mathcal{C}(\tau_2)$, and finally decodes the result to get a value of type τ_2. Creating this new function G achieves the semantics of "importing" the original function f without physically transmitting f. We extend this encoding/decoding to arbitrarily higher monotypes by inductively applying the above described process. This extension is a special case of a general technique of lifting properties on base types to higher types. For example, Leroy's [Ler92] algorithm for eliminating redundant pointer creation in a functional calculus has a similar structure. The process of generating encoder and decoder for higher-order functions can also be regarded as a systematic refinement of an *ad-hoc* method of "stub generation" for remote procedure call described in [BN84]. Since types are inferred statically, it is possible for the translator to create these encoders and decoders at compile-time.

Dealing with polymorphic functions requires mechanisms for remote type application and for creating an encoder/decoder pair at the time of type instantiation. To represent remote type application, we define the core language to have explicit type abstraction and type application. A polymorphic function is represented as a function that takes a monotype and returns a function. Remote type application can be done by a mechanism similar to remote function application by allowing *ground* monotypes to be passed through a communication network. Passing only ground types is sufficient since, under our assumption that a top level statement is a term having a closed type, type variables in a polymorphic type are fully instantiated with ground monotypes in a local site before it is executed. Since ground monotypes can be easily represented by a basic data structure, this extension does not require any special mechanism of the underlying system. To create an appropriate encoder and decoder at the time of type instantiation, we introduce two polymorphic combinators:

$$\begin{array}{lcl} \texttt{poly-encoder} & : & \forall t.t \to \mathbf{c}(t) \\ \texttt{poly-decoder} & : & \forall t.remote(\mathbf{c}(t)) \to t \end{array}$$

where $\mathbf{c}(\cdots)$ is a communication type constructor. The type system of the core language is defined in such a way that the appropriate equality holds for $\mathbf{c}$ to ensure that $\mathbf{c}(\tau) = \mathcal{C}(\tau)$ for any monotype τ. At evaluation, terms $\texttt{(poly-encoder } \delta\texttt{)}$ and $\texttt{(poly-decoder } \delta\texttt{)}$ are reduced to the encoder and decoder for ground monotype δ respectively, whose structures are described above.

We define the core language so that for any type σ, the corresponding encoder and decoder are already lambda-definable. We then develop an algorithm to translate **dML** to the core language. The algorithm must insert appropriate lambda terms representing necessary encoders/decoders of appropriate types. It also inserts type abstraction and type application for polymorphic expressions to be exported and imported properly. This process requires precise type information of expressions involving communication. The necessary type information is obtained by **dML**'s polymorphic type inference system. Thus the translation algorithm functions as follows:

1. to construct an explicitly typed term by an extended type inference algorithm, and

2. from the explicitly typed term, to generate the translated term in the core language.

Finally, we prove that the polymorphic typing is sound with respect to the semantics resulting from this translation.

4 The Core Language

The types of the core language are given by the following abstract syntax:

$$\begin{array}{l} \rho ::= t \mid b \mid \overline{b} \mid \rho \to \rho \mid id(\rho) \mid \mathbf{c}(\rho) \mid remote(\rho) \\ \pi ::= \rho \mid \forall t.\pi \mid id(\pi) \mid remote(\pi) \end{array}$$

The previous section explained $\overline{b}$, $id(\pi)$ $\mathbf{c}(\rho)$, and $remote(\pi)$. We restrict type variables of the core language to range only over the set of monotypes of **dML**. For the type constructor $\mathbf{c}$, the following equality holds:

$$\begin{array}{rcl} \mathbf{c}(b) & = & \overline{b} \\ \mathbf{c}(\tau_1 \to \tau_2) & = & id(remote(\mathbf{c}(\tau_1)) \to \mathbf{c}(\tau_2)) \end{array}$$

In what follows, we consider types as equivalence classes of this equality. A canonical representation of an equivalence class is a type satisfying the property that if it contains $\mathbf{c}(\tau)$ then τ is a type variable. Among the types, we identify the following subsets as communication types:

$$\begin{array}{l} \xi ::= \mathbf{c}(t) \mid \overline{b} \mid id(\xi \to \xi) \mid remote(\xi) \\ \mu ::= \xi \mid id(\forall t.\mu) \mid remote(\mu) \end{array}$$

Since type variables range only over monotypes of **dML**, for any communication type μ and for any **dML** monotype τ, $\mu[\tau/t]$ is also a communication type.

The core language contains the set of raw terms:

$$
\begin{array}{l}
M ::= (c : \pi) \mid x \mid \lambda x{:}\rho.M \mid \texttt{(}M\ M\texttt{)} \\
\quad \mid \lambda t.M \mid \texttt{(}M\ \tau\texttt{)} \\
\quad \mid \texttt{let } x{:}\pi \texttt{ = } e \texttt{ in } M \texttt{ end} \\
\quad \mid \texttt{makeid(M)} \mid \texttt{[}M\ M\texttt{]} \mid \texttt{[}M\ \tau\texttt{]} \\
\quad \mid \texttt{import } x{:}remote(\mu) \texttt{ in } M \texttt{ end}
\end{array}
$$

and the declarations of the form:

$$\texttt{export } M \texttt{ as } x{:}\mu$$

$(c : \pi)$ stands for typed constants, $\texttt{[}M\ M\texttt{]}$ for *remote function application*, $\texttt{[}M\ \tau\texttt{]}$ for *remote type application*, and $\texttt{makeid(}M\texttt{)}$ for identifier creation. The constants of the language include the following:

- $\texttt{(}c^b\texttt{:}b\texttt{)}$
  atomic constants,
- $\texttt{(}\overline{c^b}\texttt{:}\bar{b}\texttt{)}$
  atomic constants in canonical form,
- $\texttt{(encoder}^b\texttt{:}b \to \bar{b}\texttt{)}$
  encoders for atomic values,
- $\texttt{(decoder}^b\texttt{:}remote(\bar{b}) \to b\texttt{)}$
  decoders for atomic values,
- $\texttt{(poly-encoder:}\forall t.t \to \mathbf{c}(t)\texttt{)}$
  the polymorphic encoder, and
- $\texttt{(poly-decoder:}\forall t.remote(\mathbf{c}(t)) \to t\texttt{)}$
  the polymorphic decoder.

Figure 2 shows the set of typing rules of the core language. The first seven rules (rules from (const) through (let)) are the same as those of the explicitly typed **dML**. Rules (Rapp), (Rtapp), (import) and (export) are those for inter-process communications. Rule (Rapp) states that remote function application $\texttt{[}M_1\ M_2\texttt{]}$ is well typed if M_1 is an identifier of a function in a foreign site whose type is $remote(\xi_1) \to \xi_2$, and the second argument M_2 has the type ξ_1. The result has type $remote(\xi_2)$ and not ξ_2 itself, since it is a value of type ξ_2 sent by a foreign site. Rule (Rtapp) is for remote type application. In (export), an `export` statement is not a term but a declaration whose effect is to modify a network wide name binding. In these rules, note the restriction that some types must be communication types indicated by the usage of μ and ξ instead of π and ρ. By this restriction, the type system statically enforces the restriction that only data of communication types are being exchanged between processes. Also note that type variables in the core language ranges over monotypes of **dML**, which is indicated by the usage of τ in rules (tapp) and (Rtapp).

(const) $\mathcal{A} \rhd \texttt{(}c : \pi\texttt{)} : \pi$

(var) $\mathcal{A} \rhd x : \pi$ if $x \in domain(\mathcal{A}), \mathcal{A}(x) = \pi$

(abs) $$\frac{\mathcal{A}\{x \mapsto \rho_1\} \rhd M : \rho_2}{\mathcal{A} \rhd \lambda x.M : \rho_1 \to \rho_2}$$

(app) $$\frac{\mathcal{A} \rhd M_1 : \rho_1 \to \rho_2 \qquad \mathcal{A} \rhd M_2 : \rho_1}{\mathcal{A} \rhd \texttt{(}M_1\ M_2\texttt{)} : \rho_2}$$

(tabs) $$\frac{\mathcal{A} \rhd M : \pi}{\mathcal{A} \rhd \lambda t.M : \forall t.\pi}$$ if t not appear in $\mathcal{A}$

(tapp) $$\frac{\mathcal{A} \rhd M : \forall t.\pi}{\mathcal{A} \rhd \texttt{(}M\ \tau\texttt{)} : \pi[\tau/t]}$$

(let) $$\frac{\mathcal{A} \rhd M_1 : \pi \qquad \mathcal{A}\{x \mapsto \pi\} \rhd M_2 : \rho}{\mathcal{A} \rhd \texttt{let } x{:}\pi \texttt{ = } M_1 \texttt{ in } M_2 \texttt{ end} : \rho}$$

(makeid) $$\frac{\mathcal{A} \rhd M : \pi}{\mathcal{A} \rhd \texttt{makeid(}M\texttt{)} : id(\pi)}$$

(Rapp) $$\frac{\mathcal{A} \rhd M_1 : remote(id(remote(\xi_1) \to \xi_2)) \quad \mathcal{A} \rhd M_2 : \xi_1}{\mathcal{A} \rhd \texttt{[}M_1\ M_2\texttt{]} : remote(\xi_2)}$$

(Rtapp) $$\frac{\mathcal{A} \rhd M : remote(id(\forall t.\mu))}{\mathcal{A} \rhd \texttt{[}M\ \tau\texttt{]} : remote(\mu[\tau/t])}$$

(import) $$\frac{\mathcal{A}\{x \mapsto remote(\mu)\} \rhd M : \rho}{\mathcal{A} \rhd \texttt{import } x{:}remote(\mu) \texttt{ in } M \texttt{ end} : \rho}$$

(typeeq) $$\frac{\mathcal{A} \rhd M : \rho \qquad \rho = \rho'}{\mathcal{A} \rhd M : \rho'}$$

(export) $\texttt{export } M \texttt{ as } x{:}\mu$ is well typed if $\emptyset \rhd M : \mu$ is derivable for some closed μ.

Figure 2: The Type System of the Core Language

5 The Translation

We first develop an algorithm to translate *explicitly typed terms* of **dML** (as defined in Section 2.1) to the core language and then extend the algorithm for the implicit calculus of **dML**.

For any polytype σ of **dML**, the communication type $\mathcal{C}(\sigma)$ representing σ in the core language is defined as follows:

$$\begin{aligned} \mathcal{C}(b) &= \overline{b} \\ \mathcal{C}(t) &= \mathbf{c}(t) \\ \mathcal{C}(\tau_1 \to \tau_2) &= id(remote(\mathcal{C}(\tau_1)) \to \mathcal{C}(\tau_2)) \\ \mathcal{C}(\forall t.\sigma) &= id(\forall t.\mathcal{C}(\sigma)) \end{aligned}$$

It is easy seen that for any σ, $\mathcal{C}(\sigma)$ is a communication type.

The heart of the translation is the definitions of an encoder and decoder for each type σ in **dML**. Given a term M of type σ in the core language, we write $\Phi(\sigma, M)$ for its "encoded" term in the core language. Given a term M of type $remote(\mathcal{C}(\sigma))$ in the core language, we write $\Psi(\sigma, M)$ for the "decoded" term in the core language. Figure 3 gives the definitions for $\Phi(\sigma, M)$ and $\Psi(\sigma, M)$. It is easily seen that for any given term M in the core language, $\Phi(\sigma, M)$ and $\Psi(\sigma, M)$ are both terms in the core language. Furthermore, they have expected types:

Proposition 1 *If $\mathcal{A} \rhd M : \sigma$ is derivable in the core language then so is $\mathcal{A} \rhd \Phi(\sigma, M) : \mathcal{C}(\sigma)$.*

If $\mathcal{A} \rhd M : remote(\mathcal{C}(\sigma))$ is derivable in the core language then so is $\mathcal{A} \rhd \Psi(\sigma, M) : \sigma$.

Proof outline This is proved by showing the two statements simultaneously by induction on the structure of σ. Here we only show the cases for $\tau_1 \to \tau_2$.

For the first half of the simultaneous induction, assume $\mathcal{A} \rhd M : \tau_1 \to \tau_2$. Since $\mathcal{C}(\tau_1 \to \tau_2) = id(remote(\mathcal{C}(\tau_1)) \to \mathcal{C}(\tau_2))$ and $\mathtt{x}$ is fresh, we have only to show $\mathcal{A}\{\mathtt{x} \mapsto remote(\mathcal{C}(\tau_1))\} \rhd \Phi(\tau_2, (M\ \Psi(\tau_1, \mathtt{x})))$: $\mathcal{C}(\tau_2)$. By induction hypothesis, $\mathcal{A}\{\mathtt{x} \mapsto remote(\mathcal{C}(\tau_1))\} \rhd \Psi(\tau_1, \mathtt{x}) : \tau_1$. By the assumption on M, we have $\mathcal{A}\{\mathtt{x} \mapsto remote(\mathcal{C}(\tau_1))\} \rhd (M\ \Psi(\tau_1, \mathtt{x})) : \tau_2$. Then again by induction hypothesis, we have $\mathcal{A}\{\mathtt{x} \mapsto remote(\mathcal{C}(\tau_1))\} \rhd \Phi(\tau_2, (M\ \Psi(\tau_1, \mathtt{x})))$: $\mathcal{C}(\tau_2)$, as desired.

For the other half of the simultaneous induction, assume $\mathcal{A} \rhd M : remote(\mathcal{C}(\tau_1 \to \tau_2))$. Since $\mathtt{x}$ is fresh, by induction hypothesis we have $\mathcal{A}\{\mathtt{x} \mapsto \tau_1\} \rhd \Phi(\tau_1, \mathtt{x}) : \mathcal{C}(\tau_1)$. Since $\mathcal{C}(\tau_1 \to \tau_2) = id(remote(\mathcal{C}(\tau_1)) \to \mathcal{C}(\tau_2))$, by the assumption on M and the rule (Rtapp) we have $\mathcal{A}\{\mathtt{x} \mapsto \tau_1\} \rhd [M\ \Phi(\tau_1, \mathtt{x})] : remote(\mathcal{C}(\tau_2))$. Then again by induction hypothesis, we have $\mathcal{A}\{\mathtt{x} \mapsto \tau_1\} \rhd \Psi(\tau_2, [M\ \Phi(\tau_1, \mathtt{x})]) : \tau_2$. The desired result follows from rule (abs). ∎

Note that an explicitly typed term of **dML** is also a term in the core language. Then, using these encoders and decoders, we can define a translation algorithm that translates an explicitly typed term of **dML** to a terms of the core language. The translation algorithm is given in Figure 4. For this translation, the following property hold.

Theorem 3 *If $\mathcal{A} \rhd M : \sigma$ is derivable in the explicitly typed version of* **dML** *then $\mathcal{A} \rhd \mathcal{TR}(M) : \sigma$ is derivable in the core language.*

Proof outline This is proved by induction on the structure of M using Proposition 1. ∎

Since **dML** has a type inference algorithm that can also infer an explicitly typed term as shown in Theorems 1 and 2, a complete translation algorithm of **dML** to the core language is obtained by first applying an extended type inference algorithm to obtain an explicitly typed term and then applying the translation algorithm to get a term of the core language. Such algorithm can be easily defined as an extension of the algorithm $\mathcal{W}$. Figure 5 shows examples of the translation.

6 Operational Semantics and Soundness of Type System

This section defines an operational semantics of **dML** and establishes that the **dML** polymorphic type system is sound with respect to the semantics. Since we have an algorithm to translate **dML** to the core language, we only need to give an operational semantics of the core language. The semantics of a **dML** term is defined to be the operational semantics of the translated term of the core language.

6.1 Operational Semantics of the Core Language

The semantics is given in the style of [Tof88] by defining a set of rules to reduce terms to *canonical values* (ranged over by v). To give a precise account for communication primitives, we need to deal explicitly with the notion of multiple *sites*. We use $\alpha, \beta, \ldots$ for variables ranging over site identifiers. The set of canonical values is given by the following syntax:

$$\begin{aligned} v ::= {} & c \mid \mathtt{closure(}E\mathtt{,}T\mathtt{,}x\mathtt{,}M\mathtt{)} \\ & \mid \mathtt{Tclosure(}E\mathtt{,}T\mathtt{,}t\mathtt{,}M\mathtt{)} \mid \mathtt{id(}\alpha\mathtt{,}v\mathtt{)} \mid \mathtt{wrong} \end{aligned}$$

where c stands for the constants of the language; $\mathtt{closure(}E\mathtt{,}T\mathtt{,}x\mathtt{,}M\mathtt{)}$ is a function closure where E is a variable environment mapping a finite set of variables to canonical values and T is a type environment mapping a finite set of type variables to ground monotypes of

$$
\begin{array}{rcl}
\Phi(b, M) & = & \texttt{(encode}^b\ M\texttt{)} \\
\Phi(\tau_1 \to \tau_2, M) & = & \texttt{makeid(}\lambda\texttt{x:}\mathit{remote}(\mathcal{C}(\tau_1))\texttt{.(}\Phi(\tau_2, \texttt{(}M\ \Psi(\tau_1, \texttt{x})\texttt{))))} \quad (\texttt{x fresh}) \\
\Phi(t, e) & = & \texttt{((poly-encoder } t\texttt{) } M\texttt{)} \\
\Phi(\forall t.\sigma, M) & = & \texttt{makeid(}\lambda t\texttt{.}\Phi(\sigma, \texttt{(}M\ t\texttt{))}) \quad (t \text{ not appear elsewhere}) \\
 & & \\
\Psi(b, e) & = & \texttt{(decoder}^b\ M\texttt{)} \\
\Psi(\tau_1 \to \tau_2, M) & = & \lambda\texttt{x:}\tau_1\texttt{.(}\Psi(\tau_2, \texttt{[}M\ \Phi(\tau_1, \texttt{x})\texttt{])}) \quad (\texttt{ x fresh}) \\
\Psi(t, e) & = & \texttt{((poly-decoder } t\texttt{) } M\texttt{)} \\
\Psi(\forall t.\sigma, M) & = & \lambda t\texttt{.}\Psi(\sigma, \texttt{[}M\ t\texttt{]}) \quad (t \text{ not appear elsewhere})
\end{array}
$$

Figure 3: Definitions of Encoders and Decoders

$$
\begin{array}{rcl}
\mathcal{TR}(x) & = & x \\
\mathcal{TR}(\lambda x\texttt{:}\tau\texttt{.}M) & = & \lambda x\texttt{:}\tau\texttt{.}\mathcal{TR}(M) \\
\mathcal{TR}(\texttt{(}M_1\ M_2\texttt{)}) & = & \texttt{(}\mathcal{TR}(M_1)\ \mathcal{TR}(M_2)\texttt{)} \\
\mathcal{TR}(\lambda t\texttt{.}M) & = & \lambda t\texttt{.}\mathcal{TR}(M) \\
\mathcal{TR}(\texttt{(}M\ \tau\texttt{)}) & = & \texttt{(}\mathcal{TR}(M)\ \tau\texttt{)} \\
\mathcal{TR}(\texttt{let } x\texttt{:}\sigma\texttt{ = } M_1\texttt{ in } M_2\texttt{ end}) & = & \texttt{let } x\texttt{:}\sigma\texttt{ = } \mathcal{TR}(M_1)\texttt{ in } \mathcal{TR}(M_2)\texttt{ end} \\
\mathcal{TR}(\texttt{import } x\texttt{:}\mathit{remote}(\sigma)\texttt{ in } M\texttt{ end}) & = & \texttt{import } x\texttt{:}\mathit{remote}(\mathcal{C}(\sigma))\texttt{ in} \\
 & & \quad \texttt{let } x\texttt{:}\sigma\texttt{ = } \Psi(\sigma, x)\texttt{ in } \mathcal{TR}(M)\texttt{ end} \\
 & & \texttt{end} \\
\mathcal{TR}(\texttt{export } M\texttt{ as } x\texttt{:}\sigma) & = & \texttt{export } \Phi(\sigma, \mathcal{TR}(M))\texttt{ as } x\texttt{:}\mathcal{C}(\sigma)
\end{array}
$$

Figure 4: Translation Algorithm from **dML** to the Core Language

$\mathcal{TR}$(export λx.(x + 1) as ADDONE)
= export
makeid(λa:$remote(\overline{int})$.(encoder^{int}((λx:int.(x + 1)) (decoder^{int} a))))
as ADDONE : $id(remote(\overline{int}) \rightarrow \overline{int})$

$\mathcal{TR}$(import ADDONE:$int \rightarrow int$ in (ADDONE 3) end)
= import ADDONE:$remote(id(remote(\overline{int}) \rightarrow \overline{int}))$ in
let
addone:$int \rightarrow int$ = λa:int.(decoder^{int} [ADDONE (encoder^{int} a)])
in
(addone 3)
end
end

$\mathcal{TR}$(export λx.x as ID)
= export
makeid(λt.makeid(λa:$remote(\mathbf{c}(t))$.((poly-encoder t)
((((λt'.λx:t'.x) t) ((poly-decoder t) a)))
as ID:$id(\forall t.id(remote(\mathbf{c}(t)) \rightarrow \mathbf{c}(t)))$

$\mathcal{TR}$(import ID:$\forall t.t \rightarrow t$ in (ID 3) end)
= import ID : $remote(id(\forall t.id(remote(\mathbf{c}(t)) \rightarrow \mathbf{c}(t))))$ in
let
id = λt.λa:t.((poly-decoder t) [[ID t] ((poly-encoder t) a)])
in
((id int) 3)
end
end

Figure 5: Examples of Translation

dML; `Tclosure(`E`,`T`,`t`,`M`)` is a closure corresponding to type abstraction; $id(\alpha, v)$ is an identifier of v created in the site α; and `wrong` represents runtime type error.

We also need to assume a network wide mapping of names to values as a model of a name server. For this, we use a global variable $\mathcal{S}$ maintaining a mapping of the form $\{\mathtt{x} \mapsto v, \ldots\}$ where `x` is an identifier specified in an `export` statement and accessed in `import` statements.

We use δ as a variable ranging over the following set of ground monotypes:

$$\delta ::= \mathtt{b} \mid \delta \to \delta$$

The operational semantics is then defined by specifying the set of rules of the following form for each term constructor:

$$\alpha \vdash E, T, M \Longrightarrow v$$

which intuitively means that "in site α, M is reduced to v under the value environment E, and the type environment T." The set of reduction rules are shown in Figure 6. If none of the rule applies then the reduction terminates with `wrong`. The rules from (13) through (16) involve inter-process communication. Rules (13) and (14) involve reductions in different site, which is indicated by site id β different from α in the conclusion. Rule (15) accesses the network wide binding $\mathcal{S}$ and rule (16) modifies this binding. To emphasize that these rules involve communication, we use double lines to separate premises and the conclusion in these rules.

6.2 Soundness of the Type System

Since we have shown that the translation preserves typing (Theorem 3), the soundness of **dML** is established by showing the soundness of the polymorphic type system of the core language with respect to the operational semantics defined above.

To do this, we first define types of canonical values. Canonical values always have a closed type. We let η and γ range over *closed monotypes* and *closed polytypes* of the core language respectively. We also use the notation $\gamma[\ ]$ for a ground type with "hole"s in it and $\gamma[\tau]$ for a type obtained by replacing each hole in $\gamma[\ ]$ with τ. We write $\alpha \models v : \gamma$ if v has the type γ in site α. The set of rules to determine types of canonical values is given in Figure 7. In the rule for `Tclosure`, note that t in $\forall t.\sigma$ range only over monotypes of **dML**.

For two programs to communicate properly, they must be properly "linked" and "initialized". In our model, this requirement corresponds to requiring the property:

> for each `import` x`:`$remote(\mu)$ `in` M, there must be the corresponding export declaration `export` M `as` x`:`μ with the same type μ, and this `export` declaration must be evaluated before corresponding `import` statements are evaluated.

We assume this property.

- $\alpha \models c^{\gamma} : \gamma$
- $\alpha \models$ `closure(`E`,`T`,`x`,`M`)` $: \eta_1 \to \eta_2$
  if for any v such that $\alpha \models v : \eta_1$, if $\alpha \vdash E\{x \mapsto v\}, T, M \Longrightarrow v'$ then $\alpha \models v' : \eta_2$.
- $\alpha \models$ `Tclosure(`E`,`T`,`t`,`M`)` $: \forall t.\gamma[t]$
  if for any δ if $\alpha \vdash E, T\{t \mapsto \delta\}, M \Longrightarrow v'$ then $\alpha \models v' : \gamma[\delta]$.
- $\alpha \models$ `id(`α`,`v'`)` $: id(\gamma)$ if $\alpha \models v' : \gamma$.
- $\alpha \models v : remote(\gamma)$
  if γ is a communication type and $\beta \models v : \gamma$ for some β.

Figure 7: Types of Canonical Values

We say that an environment E *respects* a type assignment $\mathcal{A}$ in site α under a type environment T, written $E, T \models_{\alpha} \mathcal{A}$, if $domain(E) \supseteq domain(\mathcal{A})$ and for any $x \in domain(\mathcal{A})$, $T(\mathcal{A}(x))$ is a ground type and $\alpha \models E(x) : T(\mathcal{A}(x))$.

Theorem 4 *If $\mathcal{A} \rhd M : \pi$ is derivable in the core language then for any type environment T and for any environment E such that $E, T \models_{\alpha} \mathcal{A}$, if $\alpha \vdash E, T, M \Longrightarrow v$ then $\alpha \models v : T(\pi)$*

Proof outline To deal with the cases involving `import` which depends on the corresponding `export` declaration, and `poly-encoder`, `poly-decoder` combinators which dynamically generate a term and reduces it, the proof proceeds by a double induction. The outer induction is on the *import degree* defined as follows. For term `import` x`:`π `in` M, its import degree is one plus the sum of the import degrees of M and M' where M' is the term in the corresponding `export` declaration. For a term other than `import` $\cdots$, its import degree is the sum of the import degrees of its subterms. Under our assumption that programs are "properly linked" stated above, this measure is well defined. The inner induction is on the complexity measure of the lexicographical pairing of the number of `poly-encoder`s and `poly-decoder`s, and the size of the term. The proof proceeds by cases with respect to the structure of terms, using the inductive argument.

Here we only show some cases involving communications.

The case for `[`M_1 M_2`]`: assume that $\mathcal{A} \rhd$ `[`M_1 M_2`]` : $remote(\xi_1)$. By the typing rules, we must have:

(1) $\alpha \vdash E, T, x \Longrightarrow E(x) \quad \text{if } x \in \mathit{domain}(E), E(x) = v$

(2) $\alpha \vdash E, T, c \Longrightarrow c$

(3) $\alpha \vdash E, T, \lambda x{:}\rho\,.\,M \Longrightarrow \texttt{closure(}E\texttt{,}T\texttt{,}x\texttt{,}M\texttt{)}$

(4) $$\frac{\alpha \vdash E, T, M \Longrightarrow v}{\alpha \vdash E, T, \texttt{makeid(}M\texttt{)} \Longrightarrow \mathit{id}(\alpha, v)}$$

(5) $$\frac{\alpha \vdash E, T, M_1 \Longrightarrow \texttt{encoder}^b \quad \alpha \vdash E, T, M_2 \Longrightarrow c^b}{\alpha \vdash E, T, (M_1\ M_2) \Longrightarrow \overline{c^b}}$$

(6) $$\frac{\alpha \vdash E, T, M_1 \Longrightarrow \texttt{decoder}^b \quad \alpha \vdash E, T, M_2 \Longrightarrow \overline{c^b}}{\alpha \vdash E, T, (M_1\ M_2) \Longrightarrow c^b}$$

(7) $$\frac{\alpha \vdash E, T, M_1 \Longrightarrow \texttt{closure(}E'\texttt{,}T'\texttt{,}x\texttt{,}M_1'\texttt{)} \quad \alpha \vdash E, T, M_2 \Longrightarrow v \quad \alpha \vdash E'\{x \mapsto v\}, T', M_1' \Longrightarrow v'}{\alpha \vdash E, T, (M_1\ M_2) \Longrightarrow v'}$$

(8) $\alpha \vdash E, T, \lambda t\,.\,M \Longrightarrow \texttt{Tclosure(}E\texttt{,}T\texttt{,}t\texttt{,}M\texttt{)}$

(9) $$\frac{\alpha \vdash E, T, M \Longrightarrow \texttt{poly-encoder} \qquad \alpha \vdash E, T, \lambda \mathtt{x}{:}T(\tau)\,.\,\Phi(T(\tau), \mathtt{x}) \Longrightarrow v \quad (\mathtt{x} \text{ fresh})}{\alpha \vdash E, T, (M\ \tau) \Longrightarrow v}$$

(10) $$\frac{\alpha \vdash E, T, M \Longrightarrow \texttt{poly-decoder} \qquad \alpha \vdash E, T, \lambda \mathtt{x}{:}\mathit{remote}(\mathcal{C}(T(\tau)))\,.\,\Psi(T(\tau), \mathtt{x}) \Longrightarrow v \quad (\mathtt{x} \text{ fresh})}{\alpha \vdash E, T, (M\ \tau) \Longrightarrow v}$$

(11) $$\frac{\alpha \vdash E, T, M \Longrightarrow \texttt{Tclosure(}E'\texttt{,}T'\texttt{,}t\texttt{,}M'\texttt{)} \qquad \alpha \vdash E', T'\{t \mapsto T(\tau)\}, M' \Longrightarrow v}{\alpha \vdash E, T, (M\ \tau) \Longrightarrow v}$$

(12) $$\frac{\alpha \vdash E, T, M_1 \Longrightarrow v \quad \alpha \vdash E\{x \mapsto v\}, T, M_2 \Longrightarrow v'}{\alpha \vdash E, T, \texttt{let } x{:}\pi \texttt{ = } M_1 \texttt{ in } M_2 \texttt{ end} \Longrightarrow v'}$$

(13) $$\frac{\begin{array}{l}\alpha \vdash E, T, M_1 \Longrightarrow \texttt{id(}\beta\texttt{,closure(}E'\texttt{,}T'\texttt{,}x\texttt{,}M'\texttt{))} \\ \alpha \vdash E, T, M_2 \Longrightarrow v_1 \\ \beta \vdash E'\{x \mapsto v_1\}, T', M' \Longrightarrow v_2\end{array}}{\alpha \vdash E, T, [M_1\ M_2] \Longrightarrow v_2} \quad \text{if } v_1, v_2 \text{ are either of the form } \texttt{id(}\cdots\texttt{)} \text{ or } \overline{c^b}$$

(14) $$\frac{\begin{array}{l}\alpha \vdash E, T, M \Longrightarrow \texttt{id(}\beta\texttt{,Tclosure(}E'\texttt{,}T'\texttt{,}t\texttt{,}M'\texttt{))} \\ \beta \vdash E', T'\{t \mapsto T(\tau)\}, M' \Longrightarrow v\end{array}}{\alpha \vdash E, T, [M\ \tau] \Longrightarrow v} \quad \text{if } v \text{ is either of the form } \texttt{id(}\cdots\texttt{)} \text{ or } \overline{c^b}$$

(15) $$\frac{\alpha \vdash E\{x \mapsto \mathcal{S}(x)\}, T, M \Longrightarrow v}{\alpha \vdash E, T, \texttt{import } x{:}\mathit{remote}(\mu) \texttt{ in } M \texttt{ end} \Longrightarrow v}$$

(16) $$\frac{\alpha \vdash E, T, M \Longrightarrow v}{\texttt{export } M \texttt{ as } x{:}\mu \text{ is evaluated successfully at } \alpha \text{ under } E, T \text{ with the effect of } \mathcal{S} := \mathcal{S}\{x \mapsto v\}}$$

Figure 6: Operational Semantics of the Core Language (I)

$\mathcal{A} \rhd M_1 : \mathit{remote}(\mathit{id}(\mathit{remote}(\xi_2) \to \xi_1))$ and $\mathcal{A} \rhd M_2 : \xi_2$, for some communication type ξ_2. By induction hypothesis, if $\alpha \vdash E, T, M_1 \Longrightarrow v_1$ then $\alpha \models v_1 : \mathit{remote}(\mathit{id}(\mathit{remote}(T(\xi_2)) \to T(\xi_1)))$, and if $\alpha \vdash E, T, M_2 \Longrightarrow v_2$ then $\alpha \models v_2 : T(\xi_2)$. By the definition of the communication types and the typing of canonical values, the only values having a type of the form $\mathit{remote}(\mathit{id}(\mathit{remote}(T(\xi_2)) \to T(\xi_1)))$ are values of the form $\texttt{id(}\beta\texttt{,closure(}E'\texttt{,}T'\texttt{,}x\texttt{,}M'\texttt{))}$. Thus we have $v_1 = \texttt{id(}\beta\texttt{,closure(}E'\texttt{,}T'\texttt{,}x\texttt{,}M'\texttt{))}$, and $\beta \models \texttt{closure(}E'\texttt{,}T'\texttt{,}x\texttt{,}M'\texttt{)} : \mathit{remote}(T(\xi_2)) \to T(\xi_1)$. Since $\alpha \models v_2 : T(\xi_2)$, $\beta \models v_2 : \mathit{remote}(T(\xi_2))$. Then by the definition of the typing of $\texttt{closure(}\cdots\texttt{)}$, if $\beta \vdash E'\{x \mapsto v_2\}, T', M' \Longrightarrow v_3$ then $\beta \models v_3 : T(\xi_1)$, and therefore $\alpha \models v_3 : \mathit{remote}(T(\xi_1))$. Since v_2 and v_3 have a communication type, they are either $\overline{c^b}$ or a value of the form $\texttt{id(}\cdots\texttt{)}$. Then the desired property follows from reduction rule (13).

The case for $\texttt{import}\ x\texttt{:}\mathit{remote}(\mu)\ \texttt{in}\ M_1$: assume that $\mathcal{A} \rhd \texttt{import}\ x\texttt{:}\mathit{remote}(\mu)\ \texttt{in}\ M_1 : \rho$. By our assumption that terms are "properly linked" stated earlier, we must have a declaration $\texttt{export}\ M_2\ \texttt{as}\ x\texttt{:}\mu$. By the typing rules, $\emptyset \rhd M_2 : \mu$. By our assumption that programs are properly "initialized", $\texttt{export}\ M_2\ \texttt{as}\ x\texttt{:}\mu$ must be evaluated at some site β, before $\texttt{import}\ x\texttt{:}\mathit{remote}(\mu)\ \texttt{in}\ M_1$ is evaluated. By our definition, the complexity of the term M_2 is strictly smaller than that of $\texttt{import}\ x\texttt{:}\mathit{remote}(\mu)\ \texttt{in}\ M_1$. Then since $E, T \models_\beta \emptyset$, by the reduction rule (16) and the induction hypothesis, the network wide value mapping $\mathcal{S}$ satisfies the property that $\beta \models \mathcal{S}(x) : T(\mu)$. Since $T(\mu)$ is a communication type, $\alpha \models \mathcal{S}(x) : \mathit{remote}(T(\mu))$. By the typing rules, $\mathcal{A}\{x \mapsto \mathit{remote}(\mu)\} \rhd M_1 : \rho$. Then the desired property follows by the induction hypothesis and the reduction rule (15). ▮

Since `wrong` has no type, we have the following desired result.

Corollary 1 *A typechecked program in the core language does not create a run time type error.* ▮

By combining this with Theorem 3, we have the following desired property:

Corollary 2 (Soundness of dML Type System) *Suppose $\mathcal{A} \rhd e : \sigma$ is derivable in* **dML**. *Let M^e be the term of the core language obtained from e by the translation algorithm. Then for any T, for any E such that $E, T \models_\alpha \mathcal{A}$, if $\alpha \vdash E, T, M^e \Longrightarrow v$ then $\alpha \models v : T(\sigma)$.*

In particular, if e is closed, then we have $\emptyset \rhd e : \gamma$, and if $\alpha \vdash \emptyset, \emptyset, M^e \Longrightarrow v$ then $\alpha \models v : \gamma$. ▮

7 Abstract Data Types and Treatment of References

Herlihy and Liskov [HL82] considered the problem of using abstract data types remotely and proposed a solution. In their method, the programmer is required to define, for each abstract data type definition, a canonical representation together with the way data should be transmitted. Our method provides an alternative way to handle abstract data types (as well as other data structures) by exporting functions to manipulate them. The only assumption we need is that an abstract data type is implemented by a set of functions. This notion of abstract types, although somewhat different from "abstract types as existential types" as developed in [MP88], is adopted by many of the currently implemented systems including Standard ML. To extend the translation, we have only to extend our definition of $\mathcal{C}(\tau)$ to abstract data types as follows:

$$\mathcal{C}(\tau) = \mathit{id}(\tau) \text{ if } \tau \text{ is an abstract type}$$

Since values of type $\mathit{id}(\tau)$ is always an identifier, the translated abstract values can be safely sent to the remote system. This situation is the same as the case for higher-order functions.

So far we have only considered data types that have purely functional semantics. Non-functional types such as *reference types* as implemented in Standard ML [MTH90] are also very useful for some applications, and we would like to extend our method to allow remote manipulation of references. Such extension should be particularly useful for distributed programming involving large data such as dictionaries, only part of which are accessed and modified. Since a reference type can be regarded as another abstract data type specified by the operations to create, de-reference and modify a reference, it is possible to extend the core language and our translation with references based on the strategy for handling abstract data types explained above. However, it is well known that the operational semantics of references is not compatible with ML-style polymorphic type system and the type system of **dML**cannot be directly extended to references. Solutions have been proposed in [Tof88, Mac88]. They differ in details but they are both based on the idea that the type system restricts substitution on type variables in reference types in such a way that references created by a polymorphic functions have a monomorphic type. A special case of their solutions is to restrict the reference type constructor to take only a ground monotype. For this restricted case, it is fairly obvious that our method can be extended with references. We conjecture that our translation and the soundness result can also be extended to both of type systems proposed in [Tof88, Mac88]. We leave a full treatment of reference types to future investigation.

8 Application to Shared Persistent Store

One important application of our method is to develop a persistent store that can be shared by many users in a heterogeneous environment. During the research and experimentation of persistent programming languages, most notably PS-algol [ABC$^+$83], the technique to access external persistent data has been established. However, the techniques so far proposed are limited to a single language system; persistent data must be created and accessed by a single presupposed programming system. As we argued in [KO92], for persistent programming to become a viable approach to represent a shared external database system, we must break this restriction and to generalize this mechanism to a heterogeneous environment. Since our method of transparent communication allows different programming systems to exchange data, it can be used to develop such a shared persistent system.

As observed by Abadi *et.al.* [ACPP91], external data can be introduced in a static type system by the mechanism of *dynamic types*. A type system with dynamic types contains a special constant type called *Dynamic*. A runtime object of type *Dynamic* can be understood as a pair consisting of a value and its type (or more precisely a type description.) A possible operations on values of type *Dynamic* is to inspect its type component and to bind a variable of that type to its value component. Since values of type *Dynamic* contains their types, they can be safely exported to an external store and later be retrieved and used in a static type system. The recent studies [LM91, ACPR92] showed that the mechanism of dynamic types can be extended to polymorphic languages. When we combine this mechanism with our mechanism of transparent communication, then we can achieve a persistent system that can be shared in a heterogeneous environment.

In an explicitly typed version of **dML**, type *Dynamic* can be introduced by the typing rule:

$$\text{(dynamic)}\quad \frac{\mathcal{A} \rhd M : \sigma \quad (\sigma \text{ closed})}{\mathcal{A} \rhd \texttt{dynamic(}M\texttt{:}\sigma\texttt{)} : \mathit{Dynamic}}$$

As an elimination operation, instead of introducing the flexible `typecase` statement, here we only consider the following simpler coercion statement:

$$\text{(coerce)}\quad \frac{\mathcal{A} \rhd M : \mathit{Dynamic}}{\mathcal{A} \rhd \texttt{coerce(}\sigma\texttt{,}M\texttt{)} : \sigma}$$

which raises runtime exception when the type component of M is not σ. Introduction of `typecase` does not have additional difficulty.

To make values of type *Dynamic* shareable by different language systems, in the core language we restrict types of expressions that can be injected to type *Dynamic* to have a communication type. The terms `dynamic(`M`:`σ`)` and `coerce(`σ`,`M`)` are translated to the core language with this restricted *Dynamic* by the following rules:

$$\begin{aligned}&\mathcal{TR}(\texttt{dynamic(}M\texttt{:}\sigma\texttt{)})\\ &\quad = \quad \texttt{dynamic(}\Phi(\sigma, M)\texttt{:}\mathcal{C}(\sigma)\texttt{)}\\ &\mathcal{TR}(\texttt{coerce(}\sigma\texttt{,}M\texttt{)})\\ &\quad = \quad \Psi(\sigma, \texttt{coerce(}\mathit{remote}(\mathcal{C}(\sigma))\texttt{,}M\texttt{)})\end{aligned}$$

Type constructor *remote* in this context denotes the fact that the value may be created by a different language system. By this translation, a value is "encoded" and injected to type *Dynamic*, and later in a different language system it is projected to a static type and then "decoded" and used.

By combining a polymorphic type system for database programming such as the one developed in [BO92], we hope that this extension will provide a basis for representing shared database systems in a polymorphic language.

9 Conclusions and Further Investigations

We have presented a method to extend an ML-style polymorphic language with transparent communication primitives. These primitives allow programs of *any* ML types to be used remotely. Moreover, the necessary inter-process communication is completely transparent to the programmer. We have achieved this by developing a translation method for such a language to a more primitive "core" language containing only low-level inter-process communication mechanisms. We establish the *type safety* of our method by defining a precise operational semantics and proving that ML style polymorphic type system is sound with respect to the semantics. Since the core language we defined is readily implementable using a standard technique of remote procedure calls, the presented method can be incorporated in an existing statically typed language. In particular, we believe it not hard to implement a distributed extension of Standard ML by incorporating our translation algorithm in the front-end of the compiler and using some of available RPC toolkits in the runtime system. Such a system enables us to write a complex distributed application in a comfortably abstract manner. Real client-server applications require complex communication sequences, typically nested callbacks in the event-driven structure, and ML's feature of polymorphic higher-order functions should be particularly advantageous for writing such applications. Furthermore, the ability to communicate with other language

systems will open up the possibility of extending a language without modifying its compiler. For example, a language can call existing libraries written in diverse languages as far as they have suitable interface.

There are a number of ways to extend the idea presented in this paper. We conclude the paper by listing some of topics for further investigations.

- **Extensions of the Translation Scheme**
  One natural extension of our method is to extended it for communication with a language based on a different computational paradigm such as logic programming or object-oriented programming. A recent work by Lindstrom *et.al.* [LMO92] showed that by using type information it is possible to use a functional language from a logic programming. Their mechanism is mostly dynamic but has some similarity with ours in that a foreign function is passed as a form of an identifier and is called according to its type information (obtained by dynamic inspection). An interesting investigation related to their work is to ask how far can we combine their work with our static analysis and compilation? Another interesting investigation is to consider polymorphic languages with subtyping [CW85] and other object-oriented features.

- **More Efficient Translation**
  The term generated by our translation algorithm contains all the type abstraction and type application inserted by the type inference process. For implementation purpose, however, most of them are unnecessary. We can refine our translation algorithm to suppress unnecessary ones. The only necessary type abstractions and type applications are those that originate from σ in `import` $x{:}\sigma \cdots$ and `export` e `as` $x{:}\sigma$. To generate only those necessary type abstractions and type applications, we "mark", in the explicitly typed terms of **dML**, the type variables, type abstraction and type applications that are related to types in `export` and `import` statements. The type inference algorithm can be refined so that it constructs explicitly typed terms with those markings. The necessary technique is similar to that is needed to deal with eqtypes in Standard ML and are also regarded as a simple case of *kinded type abstraction* presented in [Oho92]. The translation algorithm can then be changed so that it only produces type abstractions and type applications for those marked ones.

- **More on Correctness of the Translation**
  We have proved the soundness of **dML** type system with respect to the operational semantics. This provides a strong evidence that our translation correctly achieves the desired semantics of transparent communication. It is desirable if we can also establish that the behavior of communicating programs is essentially the same as that of the corresponding single program. Intuitively, we would expect that some relationship should hold between the semantics of communicating programs of the form:

  `export` M_1 `as` x
  `import` $x{:}\sigma$ `in` M_2

  and that of the corresponding single program:

  `let` $x{:}\sigma$ `=` M_1 `in` M_2 `end`

  Since our translation algorithm may insert extra lambda abstraction which may make a nonterminating program terminating, we cannot hope that the behavior of the two be exactly same. One approach to establish a desirable relationship between the semantics of these corresponding programs might be to define a relation on the set of canonical values, and to show that if evaluation of the translated term of `let` $x{:}\sigma$ `=` M_1 `in` M_2 `end` yields a value v, then the evaluation of the translated term of `import` $x{:}\sigma$ `in` M_2 (which also involves the evaluation of the translated term of `export` M_1 `as` x) yields a value v' that is related to v.

- **Optimizations**
  A straightforward implementation of our scheme causes inter-process communication for every access to remote data. This is particularly problematic when dealing with structured data represented by abstract data types. We believe that a heuristic method can be developed to reduce communication cost. One possible approach would be to *replicate* and *cache* remote data. The first time a remote structured datum is accessed, the system sends a copy of the datum to the importing site. At the importing site, the system maintains the association of an identifier of a remote datum and the local copy of the datum. The subsequent access to any parts of the datum can then be handled locally. Since data accesses in general have locality, in many cases this may substantially improve the runtime performance. However, copying a remote datum may be problematic in such cases where the datum is very large and only small parts of it are accessed, or it contains mutable references. We regard it an important future research to develop a sound and systematic method to optimize accesses of remote structured data.

With further efforts of efficient implementation, we hope that the method presented here will provide a basis to develop a heterogeneous distributed programming system.

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), November 1983.

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transcations on Programming Languages and Systems*, 13(2):237–268, 1991.

[ACPR92] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[BMT92] D. Berry, R. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 119–129, 1992.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2(1):39–59, 1984.

[BO92] P. Buneman and A. Ohori. Polymoprhism and type inference in database programming. Technical report, University of Pennsylvania, 1992. To appear in *ACM Transaction on Database Systems*.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322, 1989.

[CK92] R. Cooper and C. Krumvieda. Distributed programming with asynchronous ordered channels in distributed ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[HS87] R. Hayes and R. D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transcations on Software Engineering*, SE-13(12):1254–1264, December 1987.

[JR86] M. B. Jones and R. F. Rashid. Mach and Matchmaker: kernel and language support for object-oriented distributed systems. In *Proceedings of ACM OOPSLA Conference*, pages 67–77, 1986.

[KO92] K. Kato and A Ohori. An approach to multilanguage persistent type system. In *Proc. Hawaii International Confernece on System Science*, pages 810–819, 1992.

[LBG+87] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. Programming Methodology Group Memo 59, MIT, 1987.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[LMO92] G. Lindstrom, J. Małuszyński, and T. Ogi. Our LISP are sealed: Interfacing functional and logic programming systems. In *Proceedings of Symposium on Programming Language Implementation and Logic Programming, Springer Lecture Notes in Computer Science, vol. 631.*, pages 428–442, 1992. August.

[Mac88] D. MacQueen. References and weak polymorphism. Note in Standard ML of New Jersey Distribution Package, 1988.

[MH88] J. C. Mitchell and R. Harper. The essence of ML. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 28–46, San Diego, California, January 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Oho92] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[Rep91] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 294–305, 1991.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988.