Distributed Meta-Programming [*]
(extended abstract)

Rui Shi
Boston University
shearer@cs.bu.edu

Chiyan Chen
Yahoo! Inc.
chiyan@yahoo-inc.com

Hongwei Xi
Boston University
hwxi@cs.bu.edu

Abstract

Distributed meta-programming (DMP), which allows code to be generated and distributed at run-time, has already become a common practice. However, code generation currently often relies on rather *ad hoc* approaches that represent code as plain text. making DMP notoriously error-prone. This unfortunate situation is further exacerbated due to issues such as mobility (of code) and locality and heterogeneity (of resources), which complicate testing and debugging drastically.

In this paper, we study distributed meta-programming from a type-theoretic perspective, presenting a language to facilitate the construction of programs that may generate and distribute code at run-time. The approach we take makes use of a form of typeful code representation developed in a previous study on meta-programming, and it guarantees statically that only well-typed code (according to some chosen type discipline) can be constructed at run-time and sent to proper locations for execution. We also mention a prototype implementation in support of the practicality of our approach to DMP, providing a solid proof of concept.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages

Keywords Distributed Meta-Programming, Typeful Code Representation, Applied Type System, ATS

1. Introduction

In this age of Internet, the need for mobile computing is ever growing. Among a variety of issues, locality and heterogeneity are of great importance in mobile computing since different locations may have different resources and/or different security concerns and thus are likely to provide different services. As a consequence, there are often needs to move entities (e.g., data, code, objects) between locations in order to access services that are only provided at certain fixed locations. For instance, a program incurring a large number of queries over a database may be executed much more efficiently if sent to the site for execution where the database is located.

We propose a typeful approach to distributed meta-programming in this paper. More specifically, we design a typed language such that the type system of the language is able to assure statically, that is, at compile-time, that code generated at run-time can only be sent to proper locations for safe execution. We choose a form of first-order abstract syntax (f.o.a.s.) based on de Bruijn indices [8] to represent typed code, which is largely adopted from some previous work on meta-programming through typeful code representation [6, 7]. The typeful code constructors are to be introduced for constructing first-order values to represent typed code, and values thus constructed can be readily sent to remote locations where the typed code represented by these values is extracted out for manipulation (including execution). In particular, if the code for a function can be represented by some first-order value, then we are able to move the function around by simply moving the value around and then executing the value to obtain the function it represents.

We use de Bruijn indices to represent free program variables in code.[1] For instance, we can declare the following datatype in ML [12] to represent pure untyped λ-expressions:

```
datatype exp =
  One | Shi of exp | Lam of exp | App of exp * exp
```

We use *One* for the first free variable in a λ-expression and *Shi* for shifting the index of each free variable in a λ-expression by one. For instance, the expression $\lambda x.\lambda y.y(x)$ can be represented as follows:

$$\mathit{Lam}(\mathit{Lam}(\mathit{App}(\mathit{One}, \mathit{Shi}(\mathit{One}))))$$

In order to represent typed expressions to be manipulated at a particular location, we refine the type *exp* into the form $\langle L, G, T \rangle$, where $\langle \cdot, \cdot, \cdot \rangle$ is a ternary type constructor and L, G, T stand for a location, a type environment (represented as a sequence of types) and a type, respectively. A value of type $\langle L, G, T \rangle$ represents some code of location L and type T in which the types of the free program variables are determined by G.[2] Therefore, the type for values representing closed code of location L and type T is $\langle L, \epsilon, T \rangle$, where ϵ stands for the empty type environment. As a simple example, let v_f be a value representing the code of some function f of type $\mathbf{int} \rightarrow \mathbf{int}$. Then v_f must be of type $\langle l, \epsilon, \mathbf{int} \rightarrow \mathbf{int} \rangle$ for some location l. The following program

$$\mathit{rget}(l, \mathit{rexec}(l, \mathit{App}(v_f, \mathit{Lift}(\mathit{enc}\ 0))))$$

[*] Partially supported by NSF grant no. CCR-0229480

[1] However, we emphasize that the use of de Bruijn indices are primarily for theoretical development. As for programming, we provide syntactic support at the source level to allow the programmer to use named variables.

[2] We use the the phrase *code of location L and type T* to mean code that can be executed (if it is closed) at location L to generate a value of type T.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010...$5.00.

essentially executes $f(0)$ remotely at location l and then fetches the result of the execution back. Both *rexec* and *rget* are primitive functions: *rexec* distributes code for remote execution and *rget* fetches values from a remote site. The built-in function *enc* encodes a (local) value v into a message, which can then be lifted by the code constructor *Lift* to form a piece of code that yields the value v when executed remotely, and the code constructor *App* forms code that does function application. A detailed explanation of various code constructors and built-in functions is to be given later.

It is certainly cumbersome, if not completely impractical, to program directly with f.o.a.s, and the use of de Bruijn indices further worsens the situation. To address this issue, we adopt some meta-programming syntax from Scheme and MetaML [18] to facilitate the construction of distributed meta-programs. The reader may take a quick look at the example in Section 3 (while ignoring the type annotations in the example for the time being).

The main contribution of the paper lies in the recognition and then formalization of a typeful approach to mobile computing through the construction of distributed meta-programs in which *code* is treated as first-class values. With its root in a previous study on meta-programming [6, 7], this approach also takes into account the issues of locality and heterogeneity in distributed programming. In particular, it makes use of a type system to guarantee that only well-typed code suitable for execution at a chosen location can actually be sent to that location for execution. In addition, we have already finished a prototype implementation (as an extension of the programming language ATS) [17] in support of the practicality of this approach, providing a solid proof of concept.

We organize the remainder of the paper as follows. In Section 2, we introduce an internal language λ_{dist} and use it as the basis for typed distributed meta-programming. We then extend λ_{dist} to λ^{+}_{dist} in Section 3, supporting the use of some meta-programming syntax in constructing distributed meta-programs, and present an example to demonstrate how distributed meta-programming can be used to support mobile computing. Lastly, we mention some related work and then conclude. Please see [5] for further details including proofs and additional examples.

2. The Language λ_{dist}

$$
\begin{array}{rlll}
\text{sorts} & \sigma & ::= & \textit{type} \mid \textit{env} \mid \textit{loc} \\
\text{types} & T & ::= & a \mid \mathbf{int} \mid T_1 \to T_2 \mid \forall a : \sigma.T \mid \exists a : \sigma.T \mid \\
 & & & \mathbf{loc}(L) \mid T@L \mid \mathbf{msg}(L,T) \mid \langle L, G, T\rangle \\
\text{c-types} & CT & ::= & \forall \vec{a} : \vec{\sigma}.(T_1, \ldots, T_n) \Rightarrow T \\
\text{type env.} & G & ::= & a \mid \epsilon \mid T :: G \\
\text{loc.} & L & ::= & a \mid \textit{here} \mid \mathbf{l}_1 \mid \mathbf{l}_2 \mid \ldots \\
\text{const.} & c & ::= & \textit{cc} \mid \textit{cf} \\
\text{exp.} & e & ::= & x \mid f \mid c(\vec{e}) \mid \mathbf{lam}\, x.\, e \mid e_1(e_2) \mid \mathbf{fix}\, f.\, e \mid \\
 & & & \forall^{+}_{\sigma}(v) \mid \forall^{-}_{\sigma}(e) \mid \exists_{\sigma}(v) \mid \\
 & & & \mathbf{let}\ \exists_{\sigma}(x) = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \\
\text{values} & v & ::= & x \mid \textit{cc}(\vec{v}) \mid \mathbf{lam}\, x.\, v \mid \forall^{+}_{\sigma}(v) \\
\text{sta. ctx.} & \Sigma & ::= & \emptyset \mid \Sigma, a : \sigma \\
\text{dyn. ctx.} & \Delta & ::= & \emptyset \mid \Delta, \textit{xf} : T
\end{array}
$$

Figure 1. The syntax for λ_{dist}

In this section, we introduce a language λ_{dist}, which essentially extends the second-order polymorphic λ-calculus with general recursion (through a fixed point operator **fix**), certain code constructors and a few special built-in functions.

2.1 Syntax

The syntax of λ_{dist} is given in Figure 1. We follow the framework *Applied Type System* ($\mathcal{ATS}$) [19] to formalize λ_{dist}. There are a static component (statics) and a dynamic component (dynamics) in λ_{dist}. Generally speaking, types are formed and reasoned about in the statics and programs are constructed and evaluated in the dynamics.

Sorts There are three sorts *type*, *env* and *loc* in the statics of λ_{dist}, and we use s for static terms, that is, terms in the statics, and a for static term variables.

- We use T for static terms of sort *type*, which serve as types for dynamic expressions e.
- We use G for static terms of sort *env*, which represent typing environments for code. For instance, ϵ represents the empty typing environment, and $T :: G$ represents the typing environment in which the type of the first program variable is T and the types of the rest of program variables are determined by G.
- We use L for static terms of sort *loc*, which simply refer to locations. In addition, we use $\mathbf{l}_1, \mathbf{l}_2, \ldots$ for constant locations and *here* for the particular location where the programmer is supposed to be at work. Therefore, if a program in λ_{dist} is executed at a location L, then *here* is interpreted to be L. In this respect, *here* is similar to I/O channels such as *stdin* and *stdout*, which are always dynamically bound.

Types In addition to the usual forms of types, λ_{dist} also supports the following ones:

- A constant type (or c-type, for short) is of the following form,
$$\forall \vec{a} : \vec{\sigma}.(T_1, \ldots, T_n) \Rightarrow T$$
where $\forall \vec{a} : \vec{\sigma}$ stands for a (possibly empty) sequence of universal quantifiers $\forall a_1 : \sigma_1 \ldots \forall a_m : \sigma_m$ and n is the arity of the constant. Note that c-types are not regarded as (regular) types. We write $c(\vec{e})$ for applying a constant c to n arguments $\vec{e} = e_1, \ldots, e_n$, where n is the arity of c. Given a constructor *cc* with arity 0, we may write *cc* for *cc*$()$.
- We may use α, γ and λ for bound variables of sorts *type*, *env* and *loc*, respectively. For instance, $\forall \alpha.T$ simply stands for the type $\forall a : \textit{type}.T[\alpha \mapsto a]$, where a is assumed to have no free occurrences in the type T.
- Given a location L, we use $\mathbf{loc}(L)$ for the singleton type such that the only value in $\mathbf{loc}(L)$ is L itself. In particular, we treat each location $\mathbf{l}_i$ as a constant of c-type $() \Rightarrow \mathbf{loc}(\mathbf{l}_i)$ in the dynamics. Also, *here* is a constant of c-type $() \Rightarrow \mathbf{loc}(\textit{here})$.
- We use $T@L$ as the type for values that serve as names for values of type T stored at location L. In some sense, a value of type $T@L$ acts like a witness attesting to some value of type T being stored at location L. The (infix) type constructor @ is abstract, and there are no typing rules associated with @. In implementation, we associate with each location L a hash table that maps names of type $T@L$, represented as integers, to values of type T stored at L.
- We use $\mathbf{msg}(L,T)$ as the type for values that can be interpreted at location L to produce values of type T at L, and call such values *messages*. A name of type $T@L$ can be turned into a message of type $\mathbf{msg}(L,T)$, but not vice versa. There exist other ways to form messages. The type constructor **msg** is abstract, and there are no typing rules associated with **msg**.
- We use $\langle L, G, T\rangle$ as the type for values representing code of location L and type T that may contain free program variables whose types are determined by G. For instance, the type $\langle L, \epsilon, T\rangle$ is for values representing *closed* code of type T that can be executed at location L. More interestingly, the type $\forall \lambda.\langle \lambda, \epsilon, T\rangle$ is for values representing closed code of type T that can be executed at every location.

$$\begin{array}{rcl}
\mathit{Lift} & : & \forall\lambda.\forall\gamma.\forall\alpha.\mathbf{msg}(\lambda,\alpha) \Rightarrow \langle\lambda,\gamma,\alpha\rangle \\
\mathit{Lam} & : & \forall\lambda.\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\lambda,\alpha_1 :: \gamma,\alpha_2\rangle) \Rightarrow \langle\lambda,\gamma,\alpha_1 \rightarrow \alpha_2\rangle \\
\mathit{App} & : & \forall\lambda.\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\lambda,\gamma,\alpha_1 \rightarrow \alpha_2\rangle, \langle\lambda,\gamma,\alpha_1\rangle) \\
 & & \quad \Rightarrow \langle\lambda,\gamma,\alpha_2\rangle \\
\mathit{Fix} & : & \forall\lambda.\forall\gamma.\forall\alpha.(\langle\lambda,\alpha :: \gamma,\alpha\rangle) \Rightarrow \langle\lambda,\gamma,\alpha\rangle \\
\mathit{One} & : & \forall\lambda.\forall\gamma.\forall\alpha.() \Rightarrow \langle\lambda,\alpha :: \gamma,\alpha\rangle \\
\mathit{Shi} & : & \forall\lambda.\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\lambda,\gamma,\alpha_1\rangle \Rightarrow \langle\lambda,\alpha_2 :: \gamma,\alpha_1\rangle
\end{array}$$

Figure 2. The typeful code constructors in λ_{dist}

$$\begin{array}{rcl}
\mathit{here} & : & () \Rightarrow \mathbf{loc}(\mathit{here}) \\
\mathit{n2m} & : & \forall\lambda.\forall\alpha.(\alpha@\lambda) \Rightarrow \mathbf{msg}(\lambda,\alpha) \\
\mathit{enc}_T & : & \forall\lambda.(T) \Rightarrow \mathbf{msg}(\lambda,T) \\
\mathit{dec} & : & \forall\alpha.(\mathbf{msg}(\mathit{here},\alpha)) \Rightarrow \alpha \\
\mathit{exec} & : & \forall\alpha.(\langle\mathit{here},\epsilon,\alpha\rangle) \Rightarrow \alpha \\
\mathit{rexec} & : & \forall\lambda.\forall\alpha.(\mathbf{loc}(\lambda), \langle\lambda,\epsilon,\alpha\rangle) \Rightarrow \alpha@\lambda \\
\mathit{get} & : & \forall\alpha.(\alpha@\mathit{here}) \Rightarrow \alpha \\
\mathit{put} & : & \forall\alpha.(\alpha) \Rightarrow \alpha@\mathit{here} \\
\mathit{rget}_T & : & \forall\lambda.(\mathbf{loc}(\lambda), T@\lambda) \Rightarrow T \\
\mathit{rput}_T & : & \forall\lambda.(\mathbf{loc}(\lambda), T) \Rightarrow T@\lambda
\end{array}$$

Figure 3. Some built-in functions in λ_{dist}

Expressions We use x for a **lam**-bound variable and f for a **fix**-bound variable, and *xf* for either an x or an f. A **lam**-bound variable is a value but a **fix**-bound variable is not. In λ_{dist}, there are also some built-in constants c, which are either constructors *cc* or functions *cf*. The marker $\forall^+_\sigma(\cdot)$ ($\forall^-_\sigma(\cdot)$) introduces (eliminates) the *universal quantification* over sorts. The $\exists_\sigma(\cdot)$ is used to handle *existential quantification*. In the following presentation, we may omit the subscript σ in a marker if it can be inferred from the context. Note that λ_{dist} imposes a form of value restriction as $\forall^+_\sigma$ is only allowed to be applied to a value.

Code Constructors The code constructors *Lift*, *One*, *Shi*, *Lam*, *App* and *Fix* are used to construct values representing typed code in which variables are replaced with de Bruijn indexes [8]. The c-types of these constructors are given in Figure 2. Unlike in meta-programming [6, 7] where *Lift* can be applied to an arbitrary value to form code, we see that *Lift* can now only be applied to a message to form code. This change is both significant and crucial as we may require that code be moved between locations in a distributed computation but can not in general assume the mobility of every value. Also, for those who are familiar with the formal language λ_{code} in [6, 7], we stress that λ_{code} *cannot* be considered a special instance of λ_{dist} because of this change.

Constant Functions We now focus on some special built-in functions in λ_{dist}, which play an indispensable role in supporting distributed meta-programming. The names and types of these functions are given in Figure 3.

- For a type T, the function enc_T takes a value v of type T to generate a message of type $\mathbf{msg}(L,T)$. A message thus generated is intended to be interpreted at location L to produce a value at L that is equivalent to v.[3] For a type T, we call enc_T the encoding function for T. We only assume the existence of encoding functions for types such as base types (e.g., **int**), location types $\mathbf{loc}(L)$, name types $T@L$, message types $\mathbf{msg}(L,T)$. In addition, we assume the existence of encoding functions for all code types, that is, types of the form $\forall\vec{a} : \vec{\sigma}.\langle L,G,T\rangle$. This is a realistic assumption as values of any code type are first-order (in the sense that they contain no functions as its subexpressions). We may omit the subscript T in enc_T if it can be readily inferred from the context.

$$\begin{array}{rcl}
\mathit{\$here} & : & \forall\lambda.() \Rightarrow \mathbf{msg}(\lambda,\mathbf{loc}(\lambda)) \\
\mathit{\$enc}_T & : & \forall\lambda_1.\forall\lambda_2.() \Rightarrow \mathbf{msg}(\lambda_1, T \rightarrow \mathbf{msg}(\lambda_2,T)) \\
\mathit{\$rexec} & : & \forall\lambda_1.\forall\lambda_2.\forall\alpha.() \Rightarrow \mathbf{msg}(\lambda_1, \langle\lambda_2,\epsilon,\alpha\rangle \rightarrow \alpha@\lambda_2) \\
\mathit{\$rget}_T & : & \forall\lambda_1.\forall\lambda_2.() \Rightarrow \mathbf{msg}(\lambda_1, \mathbf{loc}(\lambda_2) \rightarrow T@\lambda_2 \rightarrow T)
\end{array}$$

Figure 4. Some constant messages in λ_{dist}

- Given a message of type $\mathbf{msg}(\mathit{here},T)$, the function *dec* decodes the message and generates a value of type T locally. That is, *dec* dynamically translates messages to local values at runtime.
- The function *n2m* turns a name of type $T@L$ into a message of type $\mathbf{msg}(L,T)$. We can assume the existence of such a function as a name is just a remote address and thus can be readily transferred to a proper remote location to locate the actual value it refers to.
- The function *exec* is needed for executing code locally.
- Given a name referring to some local value v, the function *get* returns the value v. Given a local value v, the function *put* generates a name for v. One possibility is to use a hash table to store the mapping from names to values. Then *get* and *put* can be implemented as a lookup function and an insert function, respectively, operating on the hash table.
- The function *rexec* is needed for executing code remotely. Given a location L and a value v representing some closed code of location L and type T, *rexec* sends v to the location L to have it executed by the function *exec* that resides at L and after execution, *rexec* receives as the return result a name referring to the value of type T that is generated by the execution done at L.
- Given a location L and a name referring to some value v of type T at L, the function rget_T returns a local value equivalent to v. Given a location L and a local value v, the function *rput* puts a value v' at L that is equivalent to v and returns a name for v'. Note that we only assume the existence of rget_T and rput_T for certain types T such as **int** and code types.

Constant Messages There is an immediate need for constant messages, which we only encounter in a distributed computing environment. Suppose we have a task that needs to add two integers at a remote location L. We then need to refer to the integer addition function at L. This can be easily achieved if we have a message at hand that can be interpreted at L to obtain the integer addition function. More concretely, suppose we have a message *\$plus* of the type $\mathbf{msg}(L, \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int})$; then for two given integers i and j, the following program, where we use $\underline{i}$ and $\underline{j}$ for the messages $\mathit{enc}_{\mathbf{int}}(i)$ and $\mathit{enc}_{\mathbf{int}}(j)$, respectively, adds i and j at location L and then fetches back the result $i+j$:

$$\mathit{rget}_{\mathbf{int}}(L, \mathit{rexec}(L, \mathit{App}(\mathit{App}(\mathit{Lift}(\mathit{\$plus}), \mathit{Lift}(\underline{i})), \mathit{Lift}(\underline{j}))))$$

If *\$plus* can be interpreted at every location to obtain the integer addition function, we should then assign it the following c-type:

$$\forall\lambda.() \Rightarrow \mathbf{msg}(\lambda, \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int})$$

In the following presentation, we adopt a simple naming convention: The name of each constant message should always begin with

[3] The word *equivalent* is used here in a rather loose sense. We do not have formal definition for value equivalence at this moment, which is in general difficult to do as it requires that some form of formal semantics be assigned to values.

$$\frac{\Sigma \vdash \Delta\ [ok] \quad \Delta(xf) = T}{\Sigma; \Delta \vdash xf : T}\ \textbf{(ty-var)}$$

$$\frac{\vdash c : \forall\Sigma_0.(T_1, \ldots, T_n) \Rightarrow T \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \Delta \vdash e_i : T_i[\Theta] \text{ for } 1 \le i \le n}{\Sigma; \Delta \vdash c(e_1, \ldots, e_n) : T[\Theta]}\ \textbf{(ty-const)}$$

$$\frac{\Sigma; \Delta, x : T_1 \vdash e : T_2}{\Sigma; \Delta \vdash \textbf{lam}\, x.\, e : T_1 \to T_2}\ \textbf{(ty-lam)}$$

$$\frac{\Sigma; \Delta \vdash e_1 : T_1 \to T_2 \quad \Sigma; \Delta \vdash e_2 : T_1}{\Sigma; \Delta \vdash e_1(e_2) : T_2}\ \textbf{(ty-app)}$$

$$\frac{\Sigma; \Delta, f : T \vdash e : T}{\Sigma; \Delta \vdash \textbf{fix}\, f.\, e : T}\ \textbf{(ty-fix)}$$

$$\frac{\Sigma, a : \sigma; \Delta \vdash v : T \quad \Sigma \vdash \Delta\ [ok]}{\Sigma; \Delta \vdash \forall^+_\sigma(v) : \forall a : \sigma.T}\ \textbf{(ty-}\forall\textbf{-intro)}$$

$$\frac{\Sigma; \Delta \vdash e : \forall a : \sigma.T \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash \forall^-_\sigma(e) : T[a \mapsto s]}\ \textbf{(ty-}\forall\textbf{-elim)}$$

$$\frac{\Sigma; \Delta \vdash e : T[a \mapsto s] \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash \exists_\sigma(e) : \exists a : \sigma.T}\ \textbf{(ty-}\exists\textbf{-intro)}$$

$$\frac{\Sigma; \Delta \vdash e_1 : \exists a : \sigma.T_1 \quad \Sigma, a : \sigma; \Delta, x : T_1 \vdash e_2 : T_2}{\Sigma; \Delta \vdash \textbf{let}\ \exists_\sigma(x) = e_1\ \textbf{in}\ e_2\ \textbf{end} : T_2}\ \textbf{(ty-}\exists\textbf{-elim)}$$

Figure 5. The typing rules for λ_{dist}

the symbol \$. First, we assume that there is a constant $\$i$ of c-type $\forall\lambda.() \Rightarrow \textbf{msg}(\lambda, \textbf{int})$ for every integer i, which is needed to form a message that can be interpreted at every location to obtain the integer i. In addition, we assume the existence of some constant messages listed (partially) in Figure 4, whose meaning should be obvious. For each constant c of c-type $\forall\vec{a} : \vec{\sigma}.(T_1, \ldots, T_n) \Rightarrow T$, we use c^* to denote the function:

$$\textbf{lam}\, x_1.\, \ldots \textbf{lam}\, x_n.\, c(x_1, \ldots, x_n)$$

and $\$c$ to denote the constant message for c^*, and for $n \ge 2$, $\$^n c$ ($\$^n$ means n consecutive occurrences of \$) to denote the constant message for $\$^{n-1}c$. For instance, $\$\$get$ is assigned the following type:

$$\forall\lambda_1.\forall\lambda_2.\forall\alpha.\textbf{msg}(\lambda_1, \textbf{msg}(\lambda_2, \alpha@\lambda_2 \to \alpha))$$

Note that the primary purpose of introducing constant messages is to avoid being bothered during program construction by multiple versions of common operations like *plus* at each site.

2.2 Static Semantics

We use a judgment of the form $\Sigma \vdash s : \sigma$ to mean that the static term s can be assigned the sort σ under Σ. The rules for assigning sorts to static terms are all standard and thus omitted. We use Θ for static substitutions defined as follows,

$$\Theta \quad ::= \quad [] \mid \Theta[a \mapsto s]$$

where $[]$ stands for the empty mapping and $\Theta[a \mapsto s]$ stands for the mapping that extends Θ with a link from a to s. We assume $a \notin \textbf{dom}(\Theta)$ when writing $\Theta[a \mapsto s]$. The domain of Θ is denoted by $\textbf{dom}(\Theta)$. We write $s[\Theta]$ for the result of applying Θ to s. The standard details on substitution are all omitted. Given static contexts Σ, Σ_0 and a static substitution Θ, we write $\Sigma \vdash \Theta : \Sigma_0$ to mean $\Sigma \vdash \Theta(a) : \Sigma_0(a)$ is derivable for each $a \in \textbf{dom}(\Theta) = \textbf{dom}(\Sigma_0)$.

Given a static context Σ and a dynamic context Δ, we write $\Sigma \vdash \Delta\ [ok]$ to mean that $\Sigma \vdash \Delta(xf) : type$ is derivable for every variable *xf* in the domain $\textbf{dom}(\Delta)$ of Δ. The typing rules for λ_{dist} are listed in Figure 5, where a typing judgment is of the form $\Sigma; \Delta \vdash e : T$. One premise of the rule **(ty-const)** is $\vdash c : \forall\Sigma_0.(T_1, \ldots, T_n) \Rightarrow T$, which means that c is assigned the c-type $\forall a_1 : \sigma_1 \ldots \forall a_m : \sigma_m.(T_1, \ldots, T_n) \Rightarrow T$ for $\Sigma_0 = a_1 : \sigma_1, \ldots, a_m : \sigma_m$. Note that we have omitted some obvious side conditions associated with certain typing rules such as **(ty-∀-intro)** and **(ty-∃-elim)**.

2.3 Dynamic Semantics

As in [14], the focus of the paper is on distributed–as distinguished from concurrent– computing. The dynamic semantics of λ_{dist} is formalized in a sequential and deterministic manner. In the prototype implementation written in Objective Caml, we actually spawn a thread (at the remote execution site) whenever *rexec* is called.

We now incorporate the locality information into a typing judgment. Given a constant location L, we use $\Sigma; \Delta \vdash_L e : T$ for a typing judgment whose derivation requires the assumption that the programmer is at location L. Therefore, the old form of typing judgment $\Sigma; \Delta \vdash e : T$ now simply stands for $\Sigma; \Delta \vdash_{here} e : T$. The typing rules for the new form of typing judgments are essentially the same as those in Figure 5. However, we may assume that different sets of constants c are declared at different locations.

We also introduce a new form of expressions (and also values):

$$\begin{array}{llll} \text{exp.} & e & ::= & \ldots \mid [e]_L \\ \text{values} & v & ::= & \ldots \mid [v]_L \end{array}$$

and an additional typing rule **(ty-name)**:

$$\frac{\emptyset; \emptyset \vdash_{L'} e : T}{\Sigma; \Delta \vdash_L [e]_{L'} : T@L'}\ \textbf{(ty-name)}$$

Intuitively, $[e]_L$, which we call a remote expression, means that the expression e is to be evaluated at location L. We do not allow remote expressions to be used when constructing (source) programs in λ_{dist} because it is unclear in general as to how an (arbitrary) expression can actually be put at a remote location. According to the dynamic semantics we formulate for λ_{dist}, such a remote expression can be generated when the remote execution function *rexec* is called at run-time.

In order to assign dynamic semantics to expressions in λ_{dist}, we make use the notion of evaluation contexts defined as follows:

$$\begin{array}{llll} \text{eval. ctx.} & E & ::= & [] \mid c(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \mid \\ & & & E(e) \mid v(E) \mid \forall^-_\sigma(E) \mid \exists(E) \mid \\ & & & \textbf{let}\ \exists_\sigma(x) = E\ \textbf{in}\ e\ \textbf{end} \mid [E]_L \end{array}$$

Given a evaluation context E and an expression e, we use $E[e]$ for the expression obtained from replacing the hole $[]$ in E with e.

We define a function *comp* as follows, where we use $\overline{xfs}$ for a sequence of distinct expression variables *xf*.

$$\begin{array}{rcl} comp(\overline{xfs}; Lift(v)) & = & dec(v) \\ comp(\overline{xfs}, xf; One) & = & xf \\ comp(\overline{xfs}, xf; Shi(v)) & = & comp(\overline{xfs}; v) \\ comp(\overline{xfs}; Lam(v)) & = & \textbf{lam}\, x.\, comp(\overline{xfs}, x; v) \\ comp(\overline{xfs}; App(v_1, v_2)) & = & (comp(\overline{xfs}; v_1))(comp(\overline{xfs}; v_2)) \\ comp(\overline{xfs}; Fix(v)) & = & \textbf{fix}\, f.\, comp(\overline{xfs}, f; v) \end{array}$$

Note that *comp* is a function at meta-level. Intuitively, when applied to a sequence of distinct expression variables $\overline{xfs}$ and a value v representing some code, *comp* compiles the code to an expression. For instance, we have:

$$comp(\cdot, x, f; App(One, Shi(One))) = f(x)$$

Definition. We define redexes and their reducts as follows.

- $(\textbf{lam}\, x.\, e)(v)$ is a redex, and its reduct is $e[x \mapsto v]$.

- $\mathbf{fix}\, f.\, e$ is a redex, and its reduct is $e[f \mapsto \mathbf{fix}\, f.\, e]$.
- $\forall^-_\sigma(\forall^+_\sigma(v))$ is a redex, and its reduct is v.
- $\mathbf{let}\ \exists_\sigma(x) = \exists_\sigma(v)\ \mathbf{in}\ e\ \mathbf{end}$ is a redex, and its reduct is $e[x \mapsto v]$.
- *exec*(v) is a redex if *comp*$(\cdot; v)$ is defined, and its reduct is *comp*$(\cdot; v)$.
- *get*$([v]_{here})$ is a redex, and its reduct is v.
- *put*(v) is a redex, and its reduct is $[v]_{here}$.
- *rexec*(L, v) is a redex, and its reduct is $[\mathit{exec}(v)]_L$.
- *rget*$_T(L, [v]_L)$ is a redex, and its reduct is v.
- *rput*$_T(L, v)$ is a redex, and its reduct is $[v]_L$.
- Given a built-in function *cf* other than the above ones, *cf*$(\vec{v})$ is a redex if it is defined to be some value v, and its reduct is v. We assume that if *cf*$(\vec{v})$ can be assigned some type T, then v can also be assigned the type T. In other words, we assume that the definition of *cf* respects the c-type assigned to *cf*. Also, we expect the following to hold in the actual implementation of the built-in functions, though this cannot be enforced through types:
  - Given a value v of type T, if *enc*$_T$ is available, then *dec*$(\mathit{enc}_T(v))$ should equal v, that is, *enc*$_T$ and *dec* can be thought of as marshaling and unmarshaling operations, respectively.
  - Given a name n, *dec*$(\mathit{n2m}(n))$ should return a value v that is equivalent to the result of *get*(n).
  - For a constant message $\$c$, *dec*$(\$c)$ should return c^*.

Given expressions $e = E[e_0]$ and $e' = E[e_0']$, we write $e \hookrightarrow e'$ if e_0 is a redex and e_0' is its reduct, and say e reduces to e' in one step.

THEOREM 1 (Subject Reduction). *Assume that the typing judgment $\emptyset; \emptyset \vdash_L e : T$ is derivable and $e \hookrightarrow e'$ holds. Then the typing judgment $\emptyset; \emptyset \vdash_L e' : T$ is also derivable.*

THEOREM 2 (Progress). *Assume that the typing judgment $\emptyset; \emptyset \vdash_L e : T$ is derivable. Then either e is a value, or $e \hookrightarrow e'$ holds for some expression e', or e is of the form $E[\mathit{cf}(\vec{v})]$ such that cf$(\vec{v})$ is not a redex.*

Combining Theorem 1 and Theorem 2, we can clearly state that the evaluation of a well-typed program in λ_{dist} either reaches a value, or stops at an expression of the form $E[\mathit{cf}(\vec{v})]$ such that *cf*$(\vec{v})$ is *not* a redex, or continues forever.

However, it seems at least unwieldy, if not completely impractical, to program with abstract syntax directly. Therefore, we are naturally motivated to provide some syntactic support so as to facilitate distributed meta-programming.

3. Distributed Meta-Programming with λ^+_{dist}

We extend λ_{dist} to λ^+_{dist} with some language constructs adopted from meta-programming (supported in Scheme and MetaML):

$$\text{expr.} \quad e \quad ::= \quad \ldots \mid `(e) \mid \hat{}(e) \mid \%(e)$$

Loosely speaking, the notation $`(\cdot)$ corresponds to the back-quote notation in Scheme (or the notation $\langle\cdot\rangle$ in MetaML), and we use $`(e)$ as the code representation of e. On the other hand, $\hat{}(\cdot)$ corresponds to the comma notation in Scheme (or the notation $\tilde{}(\cdot)$ in MetaML), and we use $\hat{}(e)$ for splicing the code e into some context. We refer to $`(\cdot)$ and $\hat{}(\cdot)$ as meta-programming syntax, where $\%(\cdot)$ is a shorthand for $\hat{}(\mathit{Lift}(\cdot))$.

To facilitate the understanding of the rest of the paper, the reader may simply assume that λ^+_{dist} can be assigned a static semantics that is justified by a translation from λ^+_{dist} to λ_{dist}. In other words,

```
(* A 'withtype' clause supplies a type annotation *)

fun zeroFind f =
  let
    fun aux (i) = if f (i) = 0 then i else aux (i+1)
  in aux (0) end
withtype (int -> int) -> int

(* 'rexecInt (L, ...)' = 'rgetInt (L, rexec (L, ...))' *)
(* '{L: loc}' means universal quantification *)

fun rZeroFind1 L n = (* rpc version *)
  let
    fun f' (i: int): int =
      rexecInt (L, `(%(n2m n) %(encInt i)))
  in zeroFind f' end
withtype {L: loc} loc(L) -> (int -> int) @ L -> int

(* 'nil' for empty typing environment *)
(* 'fix f x => ...' means 'fix f. lam x => ...' *)

typedef CodeType = <L, nil, (int -> int) -> int>

fun rZeroFind2{L:loc} (L:loc(L)) (n: (int -> int) @ L)
  : int  = (* mobile code version *)
  let val zeroFindCode : CodeType =
          `(lam f =>
              (fix aux i =>
                if f (i) = 0 then i else aux (i+1)) 0)
  in rexecInt (L, `(^zeroFindCode %(n2m n))) end
```

Figure 6. RPC vs. Mobile Code

the meta-programming syntax can be treated as a form of syntactic sugar. Please see [5] for further details on the design and formalization of λ^+_{dist}.

Our concrete language for the programmer to construct distributed programs is a ML-like language. Of course, we need a process to elaborate programs written in the concrete syntax of this language into the (kind of) formal syntax of λ^+_{dist}. This is an involved process, and we unfortunately could not formally describe it and thus refer the reader to [4] for details on (partial) type inference in the framework $\mathcal{ATS}$. Instead, we are to provide some (informal) explanations and comments to facilitate the understanding of the presented code.

We present an example in Figure 6 to compare remote procedural call with mobile code. The function *zeroFind* searches for the least nonnegative zero of a function from integers to integers, and both functions *rZeroFind*$_1$ and *rZeroFind*$_2$ search for the least nonnegative zero of a remotely located function from integers to integers. Let n be a name referring to some function f at location L that maps integers to integers.

- *rZeroFind*$_1(L)(n)$ builds a function f' that serves as a local proxy for f and then calls the function *zeroFind* on f'. With this method, the amount of data/code that needs to be transmitted is unbounded and a large number of remote function calls may be invoked at run-time.
- *rZeroFind*$_2(L)(n)$ constructs the code for computing the least nonnegative zero of f and then has it executed at the location L and then fetches the result back. With this method, the amount of data/code that needs to be transmitted is bounded and low.

More interesting examples can be found in [5]. In particular, a notion of *recursive code propagation* is involved in the implementation of a peer-to-peer file search function that allows a network node to propagate the code conducting file search to its neighbors, which can in turn propagate such code to their neighbors and so on.

4. Related Work and Conclusion

There have been some recent studies that advocate the use of modal logic in designing type systems to support distributed programming [13, 10, 14], where the essential idea is to use the computational interpretation of modalities $\Box$ (necessity) and $\Diamond$ (possibility) to capture the notion of mobility/immobility in distributed computing: Given a type T, $\Box T$ is the type for mobile code of type T that can be executed everywhere, and $\Diamond T$ is the type for a name (which itself is considered to be mobile) that refers to a value of type T located somewhere.

There is a fundamental difference between λ^{+}_{dist} and the above systems based on modal logic. In λ^{+}_{dist}, a program is *entirely* located at the location *here*, namely, the place where the programmer is at work. On the other hand, it is assumed in the above systems that a program can and is also most likely to be composed of parts distributed at different locations, which may be implicit [13] or explicit [14, 10]. This may or may not be a realistic assumption depending on the actual circumstance. In the case where this is a realistic assumption, we can simply add the typing rule **(ty-name)** into λ_{dist} to support in λ_{dist} the essential features in the above systems. In particular, the modality operators $\Box$ and $\Diamond$ can be encoded as follows in λ_{dist}:

$$\Box T = \forall\lambda.\langle\lambda, \epsilon, T\rangle \qquad \Diamond T = \exists\lambda.\mathbf{loc}(\lambda) * (T@\lambda)$$

So $\Box T$ is the type for closed code of type T that can be executed everywhere, and $\Diamond T$ is the type for a pair (L, n) such that the name n can be interpreted at location L to generate a value of type T. This is largely in line with the view of modality taken in the above systems based on modal logic.

Alice [15] is a functional programming language of ML family, which shares with use some similar interests in distributed programming. In Alice, two mechanisms are provided to support distributed programming: *ticket* (reference to remote data structures) and *proxy* (reference to remote functions). Both of them relies on *pickling* that serializes values (*pickle*s) for distribution. In a loose sense, *ticket* in Alice corresponds to *name* in λ_{dist}, and *proxy* in Alice can be readily encoded through *rexec* and *rget* in λ_{dist}. As opposed to λ_{dist}, Alice's type system seems not able to statically reasoning about the *locality*.

HashML [2] is another related recent study, which mainly focuses on the dynamic nature of distributed programming. For instance, in HashML, a program value is marshaled to a string together with its type representation. Whenever the string is unmarshaled, some run-time type checking is enforced to verify that the value extracted from the string matches the type extracted from the string. HashML also needs to address issues such as run-time type names and dynamic type equality, which are not present in λ_{dist} due to different underling programming models in HashML and λ_{dist}.

Theoretical models of distributed/concurrent computation are often built upon process algebras (e.g.,[1, 11, 3, 9, 16, 20]). While we share with these works the same interest in formally studying distributed computing, there is little else in common as far as the underlying methodology is concerned. We view λ_{dist} as a novel alternative which identifies, for the first time, the close connection between distributed programming and meta-programming with a unified type theory for both of them.

In summary, we have presented a simple and general approach to support mobile computing through distributed meta-programming that combines meta-programming with distributed programming in a coherent manner. In addition, we have prototyped an implementation in support of the practicality of this approach. The implementation also immediately raises various issues such as exception handling and distributed garbage collection, which we plan to study in future.

References

[1] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[2] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-safe distributed programming for ocaml. In *Proceedings of the ACM SIGPLAN ML Workshop on ML*, Portland, Oregon, September 2006.

[3] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000. Special Issue on Coordination edited by D. Le Mtayer.

[4] C. Chen. *Type Inference in Applied Type System*. PhD thesis, Boston University, September 2005. Available at `http://cs-people.bu.edu/chiyan/thesis.html`.

[5] C. Chen, R. Shi, and H. Xi. Distributed meta-programming, 2004. Available at `http://www.cs.bu.edu/~hwxi/academic/drafts/DMP.ps`

[6] C. Chen and H. Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Uppsala, Sweden, August 2003.

[7] C. Chen and H. Xi. Meta-Programming through Typeful Code Representation. *Journal of Functional Programming*, 15(6):1–39, 2005.

[8] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes mathematicae*, 34:381–392, 1972.

[9] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, pages 406–421. Springer-Verlag LNCS 1119, 1996.

[10] L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *13th European Symposium on Programming (ESOP'04)*, pages 219–233, Barcelona, Spain, March 2004. The full paper is available as technical report no. TR-671-03 of Computer Science Department, Princeton University.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.

[12] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[13] J. Moody. Modal Logic as a Basis for Distributed Computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, 2003.

[14] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of 19th IEEE Symposium on Logic in Computer Science (LICS)*, pages 286–295, 2004.

[15] A. Rossberg, J. Schwinghammer, G. Smolka, and G. Tack. Alice Programming Language. `http://www.ps.uni-sb.de/alice/`.

[16] A. Schmitt and J.-B. Stefani. The M-calculus: a Higher-Order Distributed Process Calculus. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, pages 50–61, New Orleans, LA, January 2003.

[17] R. Shi and H. Xi. A Prototype Implementation of DMP in ATS, 2005. Available at: `http://cs-people.bu.edu/shearer/research/projects/dmp/`.

[18] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[19] H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[20] N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher-Order Processes. In *Proceedings of CONCUR'99*, pages 557–573. Springer-Verlag LNCS 1664, 1999.