

Distributed Meta-Programming*

Chiyan Chen Rui Shi Hongwei Xi

Boston University

{chiyan, shearer, hwxi}@cs.bu.edu

October 5, 2004

Abstract

The need for mobile computing is evident and growing rapidly in this age of Internet. However, it is in general difficult to construct programs in support of mobile computing due to issues such as mobility (of code) and locality and heterogeneity (of resources). 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. In particular, we are to form a type system that can statically guarantee that only well-typed code (according to some chosen type discipline) can be constructed and then sent to a proper location for execution. To achieve this, we make use of a form of typeful code representation developed in a previous study on meta-programming. The main contribution of the paper lies in the recognition and then formalization of a typeful approach to distributed meta-programming that is simple as well as general. In addition, we have finished a prototype implementation in support of the practicality of this approach, providing a solid proof of concept.

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

* Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

1 Introduction

In this age of Internet, the need for mobile computing is growing rapidly. 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. It is in such a context that questions about mobility/immobility of entities are raised naturally.

We can hide the issue of location as much as possible by relying on certain implementation strategies. For instance, we can use various marshalling/unmarshalling functions to move data of certain types (e.g., various base types) between locations and then provide proper local interfaces (a.k.a. proxies) to support remote procedural calls. With such a design, functions are often not treated as data that can be moved between locations. In general, there are various reasons that make it problematic to move functions between locations. First and foremost, it is semantically rather unclear as to what should really happen when a function, which may cause effects, is moved from one location to another. However, entirely prohibiting functions from being moved around may make mobile computing too restricted as it does make sense, sometimes, to move certain functions between locations. For instance, a function 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.

In this paper, we design a language to support distributed meta-programming that allows code to be generated at run-time and then sent to remote locations for execution. In order to guarantee that only well-typed code (according to some chosen type discipline) can ever be generated, we are to make use of a form of typeful representation for code, that is, a form of representation that allows the type of code to be reflected in the type of the value that represents the code. In addition, we are to make sure that only closed code, that is, code containing no free program variables, can actually be allowed for execution.

A common approach to capturing the notion of closed code is through higher-order abstract syntax (h.o.a.s.) (Church 1940; Pfenning and Elliott 1988; Pfenning). Though clean and elegant, there are some fundamental problems with representing code as h.o.a.s. trees. Firstly, the essence of h.o.a.s. is to use functions in the meta-language to represent functions in the object-language. Therefore, if there is a mismatch between the function space in the meta-language and the one in the object-language, code representation through h.o.a.s. cannot be adequate. Secondly, it seems rather difficult, if not impossible, to manipulate open code, that is, code containing free program variables, in a satisfactory manner when higher-order code representation is chosen, but we may encounter cases where there is a need for manipulating open code directly. Thirdly, we need to send code to remote locations for execution but this can be difficult to do with higher-order code representation.

We choose a form of first-order abstract syntax trees to represent typed code, which is largely adopted from some previous work on meta-programming through typeful code representation (Chen and Xi 2003); 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 a remote location; the typed code represented by these values can then be extracted at the remote location 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.

We are to use deBruijn indexes (de Bruijn 1972) to represent free program variables in code. For instance, we can declare the following datatype in Standard ML 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 each free variable in a λ -expression by one index. As an example, the expression $\lambda x.\lambda y.y(x)$ can be represented as follows:

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

For representing typed expressions that are 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^1 in which the types of the free program variables are determined by *G*. 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.

It is certainly cumbersome, if not completely impractical, to program with f.o.a.s. trees, and the direct use of deBruijn indexes further worsens the situation. To address this issue, we adopt some meta-programming syntax from Scheme and MetaML (Taha and Sheard 2000) to facilitate the construction of distributed meta-programs and then provide a translation to eliminate the meta-programming syntax. We also provide interesting examples in support of this design.

The main contribution of the paper lies in the recognition and then formalization of a novel approach to mobile computing through the construction of distributed meta-programs. With its root in a previous study on meta-programming through typeful code representation, 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.

We believe that we have presented for the first time a type-theoretic account for distributed meta-programming in which code is treated as first-class values. In addition, we have so far already finished a prototype implementation 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. In Section 4, we argue that the presented approach to distributed meta-programming can be readily extended to support common programming features. We also present

¹We 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*.

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

 Figure 1: The syntax for λ_{dist}

some examples in Section 5 to demonstrate how distributed meta-programming can be used to support mobile computing. Lastly, we mention some related work and then conclude.

The paper is largely self-contained, and if needed, further examples and explanations of typeful code representation can be found in (Chen and Xi 2003; Chen and Xi 2004). We will also point out some crucial changes that we need to adopt in order to make use of typeful code representation in a distributed computing environment.

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

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

2 The Language 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 \mathbf{fix}), certain code constructors and a few special built-in functions. The syntax of λ_{dist} is given in Figure 1.

We follow the framework *Applied Type System (ATS)* (Xi 2004) 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

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

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

the dynamics. 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 $\mathbf{l}_1, \mathbf{l}_2, \dots$ for constant locations.

We use *here* to refer to 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 like *stdin* and *stdout*, which are always dynamically bound.

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

- 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 $() \Rightarrow \mathbf{loc}(\mathit{here})$.
- 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 in general. The type constructor \mathbf{msg} is abstract, and there are no typing rules associated with \mathbf{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 ; the type $\forall a : \text{loc}. \langle a, \epsilon, T \rangle$ is for values representing closed code of type T that can be executed at every location.

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 . Each of these constants is assigned a constant type (or c-type, for short) of the following form,

$$\forall \vec{a} : \vec{\sigma}. (T_1, \dots, T_n) \Rightarrow T$$

where n is the arity of the constant. We use $\forall \vec{a} : \vec{\sigma}$ for a (possibly empty) sequence of universal quantifiers $\forall a_1 : \sigma_1 \dots \forall a_m : \sigma_m$. Note that c-types are not regarded as (regular) types. We write $c(\vec{e})$ for applying a constant c to n arguments $\vec{e} = e_1, \dots, e_n$, where n is the arity of c . We may write cc for $cc()$ when the arity of a constructor cc is 0. Also, we may use α , γ and λ for variables of sorts *type*, *env* and *loc*, respectively. For instance, $\forall \alpha. T$ simply stands for the type $\forall a : \text{type}. T[\alpha \mapsto a]$, where a is assumed to have no free occurrences in the type T .

The markers $\forall_\sigma^+(\cdot)$, $\forall_\sigma^-(\cdot)$ and $\exists_\sigma(\cdot)$ are introduced to allow that the last rule applied in the typing derivation of an expression e be uniquely determined by the structure of e . This in turn makes it significantly easier to establish Theorem 2.4 (subject reduction) and Theorem 2.5 (progress) for λ_{dist} . 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_σ^+ is only allowed to be applied to a value.

The code constructors *Lift*, *One*, *Shi*, *Lam*, *App* and *Fix* are to be used for constructing values representing typed code in which variables are replaced with deBruijn indexes (de Bruijn 1972), and the c-types of these constructors are given in Figure 2. For those who are unfamiliar with such constructors, it can be helpful to first study some related meta-programming examples (Chen and Xi 2003). Unlike in meta-programming 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 needed as we may require that code be moved between locations in a distributed computing but can not in general assume the mobility of every value. Also, for those familiar with λ_{code} in (Chen and Xi 2003), we stress that λ_{code} cannot be considered a special instance of λ_{dist} because of this change.

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

- For certain closed types T , the functions enc_T take 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 .² For each type T , we call enc_T the encoding function for T . We assume the existence of encoding functions for types such as base types (e.g., **int**), location types **loc**(L), name types $T@L$, message types **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.

- Given a message of type **msg**(*here*, T), the function *dec* decodes the message and generates a value of type T locally.
- The function *n2m* turns a name of type $T@L$ into a message of type **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. To facilitate understanding, we present some *untyped* code as follows in ML-like syntax, which illustrates a possible implementation of *exec*:

```

fun eval (p, env) = (* 'env' is a list of values *)
  case p of
    Lift msg => decode msg (* interpret a message *)
  | One => hd (env)
  | Shi p => eval (p, tl (env))
  | Lam (p) => lam x => eval (p, x :: env)
  | App (p1, p2) => eval (p1, env) (eval (p2, env))
  | Fix (p) => fix (lam f => eval (p, f :: env))

fun exec p = eval (p, [])

```

We use *fix* here as a function that computes the fixed point of its argument. Also, we emphasize that for the sake of efficiency, an implementation of *exec* in practice may compile its argument, which represents some closed code, before execution.

- 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

²The 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 some form of formal semantics must be assigned to values.

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.
- Given a type T such that $rget_T$ exists, we use $rexec_T$ for a function that is essentially equivalent to the following function:

$$\mathbf{lam} \ l. \mathbf{lam} \ x. rget_T(l, rexec(l, x))$$

However, $rexec_T$ may support a more efficient implementation.

As an example, the function $rget_T$ for type $T = T_1 \rightarrow T_2$ can essentially be implemented as follows,

$$\mathbf{lam} \ l. \mathbf{lam} \ x. \mathbf{lam} \ x_1. rget_{T_2}(l, rexec(l, App(Lift(n2m(x)), Lift(n2m(rput_{T_1}(l, x_1)))))))$$

where the availability of both $rput_{T_1}$ and $rget_{T_2}$ is assumed. In this implementation, $rget_T$ does not actually fetch a function from a remote location. It instead builds a proxy locally for calling the remote function. In the case where both T_1 and T_2 are base types, the argument of such a call is literally put to the remote location and the result of the call is literally fetched back.

We consider it a key contribution of the paper to select the special functions in Figure 3 and then assign them proper types, which requires considerable technical insights. We suggest that the reader gain a clear understanding of these special functions and their types before studying the subsequent development.

$$\begin{aligned}
 \$here & : \forall \lambda. () \Rightarrow \mathbf{msg}(\lambda, \mathbf{loc}(\lambda)) \\
 \$n2m & : \forall \lambda_1. \forall \lambda_2. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda_1, \alpha @ \lambda_2 \rightarrow \mathbf{msg}(\lambda_2, \alpha)) \\
 \$enc_T & : \forall \lambda_1. \forall \lambda_2. () \Rightarrow \mathbf{msg}(\lambda_1, T \rightarrow \mathbf{msg}(\lambda_2, T)) \\
 \$dec & : \forall \lambda. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda, \mathbf{msg}(\lambda, \alpha) \rightarrow \alpha) \\
 \$exec & : \forall \lambda. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda, \langle \lambda, \epsilon, \alpha \rangle \rightarrow \alpha) \\
 \$get & : \forall \lambda. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda, \alpha @ \lambda \rightarrow \alpha) \\
 \$put & : \forall \lambda. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda, \alpha \rightarrow \alpha @ \lambda) \\
 \$rexec & : \forall \lambda_1. \forall \lambda_2. \forall \alpha. () \Rightarrow \mathbf{msg}(\lambda_1, \langle \lambda_2, \epsilon, \alpha \rangle \rightarrow \alpha @ \lambda_2) \\
 \$rget_T & : \forall \lambda_1. \forall \lambda_2. () \Rightarrow \mathbf{msg}(\lambda_1, \mathbf{loc}(\lambda_2) \rightarrow T @ \lambda_2 \rightarrow T) \\
 \$rput_T & : \forall \lambda_1. \forall \lambda_2. () \Rightarrow \mathbf{msg}(\lambda_1, \mathbf{loc}(\lambda_2) \rightarrow T \rightarrow T @ \lambda_2) \\
 \$rexec_T & : \forall \lambda_1. \forall \lambda_2. () \Rightarrow \mathbf{msg}(\lambda_1, \langle \lambda_2, \epsilon, T \rangle \rightarrow T)
 \end{aligned}$$

Figure 4: Some constant messages in λ_{dist}

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 $enc_{\mathbf{int}}(i)$ and $enc_{\mathbf{int}}(j)$, respectively, adds i and j at location L and then fetches back the result $i + j$:

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

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 the symbol $\$$. First, we assume that there is a constant $\$i$ of c-type $\forall \lambda. () \Rightarrow \mathbf{msg}(\lambda, \mathbf{int})$ 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 the constant messages listed in Figure 4, whose meaning should be obvious. Given a value v , we say that c is a constant message for v if the interpretation of c by the function dec yields the value v . In general, if a value is assigned the type $\forall \vec{a} : \vec{\sigma}. T$, where T does not begin with a universal quantifier, then the constant message for the value, if it exists, should be given the following type: $\forall \lambda. \forall \vec{a} : \vec{\sigma}. \mathbf{msg}(\lambda, T[here \mapsto \lambda])$, where λ is assumed to have no free occurrence in T . Note that the substitution $[here \mapsto \lambda]$ clearly reflects the nature of dynamic binding imposed on $here$. For instance, it is clearly stated in the type of $\$here$ that the value L (not $here$) should be returned if $\$here$ is interpreted at location L (since L is $here$ at location L).

For each constant c of c-type $\forall \vec{a} : \vec{\sigma}. (T_1, \dots, T_n) \Rightarrow T$, we use c^* to denote the following function,

$$\mathbf{lam} x_1. \dots \mathbf{lam} x_n. c(x_1, \dots, x_n)$$

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

$$\forall \lambda_1. \forall \lambda_2. \forall \alpha. \mathbf{msg}(\lambda_1, \mathbf{msg}(\lambda_2, \alpha @ \lambda_2 \rightarrow \alpha))$$

As another example, we can define $rput_T$ as follows:

$$\begin{aligned} &\mathbf{lam} l. \mathbf{lam} x. \\ &\quad rexec(l, \\ &\quad\quad App(App(Lift \$rget_T, Lift(enc(here))), Lift(enc(put x)))) \end{aligned}$$

where the first *enc* is $enc_{\text{loc}(\text{here})}$ and the second one is $enc_{T@\text{here}}$. Let us now see what happens if we want to put a value v of type T at a location L with this implementation; we first use *put* to create a name for v and then execute at location L some code that essentially relies on the $rget_T$ function at L to form a value v' at L equivalent to v by using the name for v . In other words, putting a value of type T at location L is achieved by calling $rget_T$ at location L on a name generated for v .

Typing rules $\Sigma; \Delta \vdash e : T$

$$\begin{array}{c}
 \frac{\Sigma \vdash \Delta \text{ [ctx] } \quad \Delta(xf) = T}{\Sigma; \Delta \vdash xf : T} \text{ (ty-var)} \\
 \frac{\vdash c : \forall \Sigma_0. (T_1, \dots, T_n) \Rightarrow T \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \Delta \vdash e_i : T_i[\Theta] \text{ for } 1 \leq i \leq n}{\Sigma; \Delta \vdash c(e_1, \dots, e_n) : T[\Theta]} \text{ (ty-const)} \\
 \frac{\Sigma; \Delta, x : T_1 \vdash e : T_2}{\Sigma; \Delta \vdash \mathbf{lam} \ x. e : T_1 \rightarrow T_2} \text{ (ty-lam)} \\
 \frac{\Sigma; \Delta \vdash e_1 : T_1 \rightarrow T_2 \quad \Sigma; \Delta \vdash e_2 : T_1}{\Sigma; \Delta \vdash e_1(e_2) : T_2} \text{ (ty-app)} \\
 \frac{\Sigma; \Delta, f : T \vdash e : T}{\Sigma; \Delta \vdash \mathbf{fix} \ f. e : T} \text{ (ty-fix)} \\
 \frac{\Sigma, a : \sigma; \Delta \vdash v : T \quad \Sigma \vdash \Delta \text{ [ctx]}}{\Sigma; \Delta \vdash \forall_{\sigma}^+(v) : \forall a : \sigma. T} \text{ (ty-}\forall\text{-intro)} \\
 \frac{\Sigma; \Delta \vdash e : \forall a : \sigma. T \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash \forall_{\sigma}^-(e) : T[a \mapsto s]} \text{ (ty-}\forall\text{-elim)} \\
 \frac{\Sigma; \Delta \vdash e : T[a \mapsto s] \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash \exists_{\sigma}(e) : \exists a : \sigma. T} \text{ (ty-}\exists\text{-intro)} \\
 \frac{\Sigma; \Delta \vdash e_1 : \exists a : \sigma. T_1 \quad \Sigma, a : \sigma; \Delta, x : T_1 \vdash e_2 : T_2}{\Sigma; \Delta \vdash \mathbf{let} \ \exists_{\sigma}(x) = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end} : T_2} \text{ (ty-}\exists\text{-elim)}
 \end{array}$$

Figure 5: The typing rules for λ_{dist}

2.1 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,

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

and $\mathbf{dom}(\Theta)$ for the domain of Θ . Note that $[]$ stands for the empty mapping and $\Theta[a \mapsto s]$ stands for the mapping that extends Θ with a link from a to s , where $a \notin \mathbf{dom}(\Theta)$ is assumed. 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 $a \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma_0)$.

Given a static context Σ and a dynamic context Δ , we write $\Sigma \vdash \Delta [ok]$ to mean that $\Sigma \vdash \Delta(x) : type$ is derivable for every variable xf in the domain $\mathbf{dom}(\Delta)$ 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 $\vdash c : \forall \Sigma_0. (T_1, \dots, T_n) \Rightarrow T$, which means that c is assigned the c -type $\forall a_1 : \sigma_1 \dots \forall a_m : \sigma_m. (T_1, \dots, T_n) \Rightarrow T$ for $\Sigma_0 = a_1 : \sigma_1, \dots, a_m : \sigma_m$. Note that we have omitted some obvious side conditions associated with certain typing rules such as **(ty- \forall -intro)** and **(ty- \exists -elim)**.

Proposition 2.1 (Canonical Forms) *Assume that $\emptyset; \emptyset \vdash v : T$ is derivable.*

- If T is of the form $\mathbf{loc}(L)$, then v is L .
- If T is of the form $T_1 \rightarrow T_2$, then v is of the form $\mathbf{lam} x. e$.
- If T is of the form $\langle L, G, T_0 \rangle$, then v is of the form $\mathbf{Lift}(v_0)$, \mathbf{One} , $\mathbf{Shi}(v_0)$, $\mathbf{Lam}(v_0)$, $\mathbf{App}(v_1, v_2)$, or $\mathbf{Fix}(v_0)$.
- If T is of the form $\forall a : \sigma. T_0$, then v is of the form $\forall_\sigma^+(v_0)$.
- If T is of the form $\exists a : \sigma. T_0$, then v is of the form $\exists_\sigma(v_0)$.

Proof By a careful inspection of the typing rules in Figure 5. ■

2.2 Dynamic Semantics

As in (Murphy, Crary, Harper, and Pfenning 2004), 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 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:

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

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

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

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. We will come back to this point when discussing related work in Section 6. However, 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.

We use θ for dynamic substitutions defined as follows,

$$\theta ::= [] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e]$$

and write $e[\theta]$ for the result of applying θ to e . We write

$$\Sigma; \Delta \vdash (\Theta; \theta) : (\Sigma_0; \Delta_0)$$

to mean that $\Sigma \vdash \Theta : \Sigma_0$ holds and for each $xf \in \mathbf{dom}(\theta) = \mathbf{dom}(\Delta_0)$, $\Sigma; \Delta \vdash \theta(xf) : \Delta_0(xf)[\Theta]$ is derivable.

Lemma 2.2 (Substitution) *Assume that the typing judgment $\Sigma, \Sigma_0; \Delta, \Delta_0 \vdash e : T$ is derivable and $\Sigma; \Delta \vdash (\Theta; \theta) : (\Sigma_0; \Delta_0)$ holds. Then $\Sigma; \Delta \vdash e[\theta] : T[\Theta]$.*

Proof The proof follows from structural induction on the typing derivation of $\Sigma, \Sigma_0; \Delta, \Delta_0 \vdash e : T$. ■

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

$$\begin{aligned} \text{eval. ctx. } E ::= & \\ & [] \mid c(v_1, \dots, v_{i-1}, E, e_{i+1}, \dots, e_n) \mid E(e) \mid v(E) \mid \\ & \forall_{\sigma}(E) \mid \exists(E) \mid \mathbf{let} \exists_{\sigma}(x) = E \mathbf{in} e \mathbf{end} \mid [E]_L \end{aligned}$$

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{xf} s for a sequence of distinct expression variables xf .

$$\begin{aligned} \mathit{comp}(\overline{xf}s; \mathit{Lift}(v)) &= \mathit{dec}(v) \\ \mathit{comp}(\overline{xf}s; \mathit{One}) &= \mathit{xf} \\ \mathit{comp}(\overline{xf}s; \mathit{Shi}(v)) &= \mathit{comp}(\overline{xf}s; v) \\ \mathit{comp}(\overline{xf}s; \mathit{Lam}(v)) &= \mathbf{lam} x. \mathit{comp}(\overline{xf}s, x; v) \\ \mathit{comp}(\overline{xf}s; \mathit{App}(v_1, v_2)) &= (\mathit{comp}(\overline{xf}s; v_1))(\mathit{comp}(\overline{xf}s; v_2)) \\ \mathit{comp}(\overline{xf}s; \mathit{Fix}(v)) &= \mathbf{fix} f. \mathit{comp}(\overline{xf}s, f; v) \end{aligned}$$

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

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

Definition 2.3 We define redexes and their reductions as follows.

- $(\mathbf{lam} x. e)(v)$ is a redex, and its reduction is $e[x \mapsto v]$.
- $\mathbf{fix} f. e$ is a redex, and its reduction is $e[f \mapsto \mathbf{fix} f. e]$.
- $\forall_{\sigma}^{-}(\forall_{\sigma}^{+}(v))$ is a redex, and its reduction is v .
- $\mathbf{let} \exists_{\sigma}(x) = \exists_{\sigma}(v) \mathbf{in} e \mathbf{end}$ is a redex, and its reduction is $e[x \mapsto v]$.
- $\mathbf{exec}(v)$ is a redex if $\mathbf{comp}(\cdot; v)$ is defined, and its reduction is $\mathbf{comp}(\cdot; v)$.
- $\mathbf{get}([v]_{\mathit{here}})$ is a redex, and its reduction is v .
- $\mathbf{put}(v)$ is a redex, and its reduction is $[v]_{\mathit{here}}$.
- $\mathbf{rexec}(L, v)$ is a redex, and its reduction is $[\mathbf{exec}(v)]_L$.
- $\mathbf{rget}_T(L, [v]_L)$ is a redex, and its reduction is v .
- $\mathbf{rput}_T(L, v)$ is a redex, and its reduction 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 reduction 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 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 \mathbf{enc}_T is available, then $\mathbf{dec}(\mathbf{enc}_T(v))$ should equal v , that is, \mathbf{enc}_T and \mathbf{dec} can be thought of as marshalling and unmarshalling.
 - Given a name n , $\mathbf{dec}(n2m(n))$ should return a value v that is equivalent to the result of $\mathbf{get}(n)$.
 - For a constant message $\$c$, $\mathbf{dec}(\$c)$ 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 reduction, and say e reduces to e' in one step.

Theorem 2.4 (Subject Reduction) Assume that the typing judgment $\emptyset; \emptyset \vdash e : T$ is derivable and $e \hookrightarrow e'$ holds. Then the typing judgment $\emptyset; \emptyset \vdash e' : T$ is also derivable.

Proof Assume that $e = E[e_0]$ and $e' = E[e'_0]$ for some redex e_0 and its reduction e'_0 . The proof proceeds by induction on E , and the most interesting case is where $E = []$. In this case, the proof proceeds by induction on the typing derivation of $\emptyset; \emptyset \vdash e : T$. ■

Theorem 2.5 (Progress) Assume that the typing judgment $\emptyset; \emptyset \vdash e : T$ is derivable. Then either e is a value, or $e \hookrightarrow e'$ holds for some expression e' , or e is of the form $E[cf(\vec{v})]$ such that $cf(\vec{v})$ is not a redex.

Proof This follows from structural induction on the typing derivation of $\emptyset; \emptyset \vdash e : T$. ■

Combining Theorem 2.4 and Theorem 2.5, 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 $E[cf(\vec{v})]$ such that $cf(\vec{v})$ is *not* a redex, or continues forever.

In theory, it is already possible to do distributed meta-programming with λ_{dist} . As an example, we show how remote procedure calls (RPCs) can be performed in λ_{dist} . Let n be a name of type $(T_1 \rightarrow T_2)@L$, that is, n refers to a function of type $T_1 \rightarrow T_2$ located remotely at L . If there is an encoding function enc_{T_1} for the type T_1 , then calling the function referred to by n with a local value v of type T_1 can be performed by the following program:

$$rexec(L, App(Lift(n2m(n)), Lift(enc_{T_1}(v))))$$

which yields a name referring to the value at location L that is returned by the remote function call. The significance of this example should be noticed: Instead of relying on a language primitive to perform RPCs, we have made RPCs implementable in λ_{dist} . As can be expected, we are now able to implement much more than just RPCs.

However, it seems at least unwieldy, if not completely impractical, to program with abstract syntax trees directly. To some extent, this is like constructing meta-programs in Scheme with no access to the backquote/comma notation. Therefore, we are naturally motivated to provide some syntactic support so as to facilitate distributed meta-programming. This is precisely the situation we encounter in meta-programming (Chen and Xi 2003).

$$\begin{aligned}
 Lift_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha. (\mathbf{msg}(\lambda_1, \mathbf{msg}(\dots \mathbf{msg}(\lambda_n, \alpha) \dots))) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha \rangle \dots \rangle \rangle \\
 One_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha. () \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \alpha :: \gamma_n, \alpha \rangle \dots \rangle \rangle \\
 Shi_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha_1. \forall \alpha_2. (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha_1 \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \alpha_2 :: \gamma_n, \alpha_1 \rangle \dots \rangle \rangle \\
 Lam_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha_1. \forall \alpha_2. (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \alpha_1 :: \gamma_n, \alpha_2 \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha_1 \rightarrow \alpha_2 \rangle \dots \rangle \rangle \\
 App_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha_1. \forall \alpha_2. \\
 & \quad (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha_1 \rightarrow \alpha_2 \rangle \dots \rangle \rangle, \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha_1 \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha_2 \rangle \dots \rangle \rangle \\
 Fix_n & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall \alpha. \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \alpha :: \gamma_n, \alpha \rangle \dots \rangle \rangle \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \alpha \rangle \dots \rangle \rangle
 \end{aligned}$$

Figure 6: The generalized typeful code constructors in λ_{dist}^+

3 The Language λ_{dist}^+

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

$$\text{expressions } e ::= \dots \mid '(e) \mid \hat{(}e)$$

Loosely speaking, the notation $'(\cdot)$ corresponds to the backquote notation in Scheme (or the notation $\langle \cdot \rangle$ in MetaML), and we use $'(e)$ as the code representation for e . On the other hand, $\hat{(\cdot)}$ corresponds to the comma notation in Scheme (or the notation $\sim(\cdot)$ 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.

Typing rules $\Sigma; \Gamma \vdash_{\mathcal{L}}^{\mathcal{G}} e : T$

$$\begin{array}{c}
 \frac{\Sigma \vdash_{\mathcal{L}}^{\mathcal{G}} \Delta [ok] \quad \Delta(xf@|\mathcal{L}|) = T}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} xf : T} \text{ (ty-var)} \\
 \\
 \frac{\Sigma(c) = \forall \Sigma_0. (T_1, \dots, T_n) \Rightarrow T \quad \Sigma \vdash_{\mathcal{L}}^{\mathcal{G}} \Delta [ok] \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e_i : T_i[\Theta] \text{ for } i = 1, \dots, n}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} c(e_1, \dots, e_n) : T[\Theta]} \text{ (ty-const)} \\
 \\
 \frac{\Sigma; \Delta, x@|\mathcal{L}| : T_1 \vdash_{\mathcal{L}}^{\mathcal{G}} e : T_2}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} \mathbf{lam } x. e : T_1 \rightarrow T_2} \text{ (ty-lam)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e_1 : T_1 \rightarrow T_2 \quad \Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e_2 : T_1}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e_1(e_2) : T_2} \text{ (ty-app)} \\
 \\
 \frac{\Sigma; \Delta, f@|\mathcal{L}| : T \vdash_{\mathcal{L}}^{\mathcal{G}} e : T}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} \mathbf{fix } f. e : T} \text{ (ty-fix)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\mathcal{L}+L}^{\mathcal{G}+G} e : T}{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} \langle e \rangle : \langle L, G(|\mathcal{L}| + 1; \Delta), T \rangle} \text{ (ty-encode)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e : \langle L, G(|\mathcal{L}| + 1; \Delta), T \rangle}{\Sigma; \Delta \vdash_{\mathcal{L}+L}^{\mathcal{G}+G} \hat{\langle e \rangle} : T} \text{ (ty-decode)} \\
 \\
 \frac{\Sigma, a : \sigma; \Delta \vdash_{\emptyset}^{\emptyset} v : T \quad \Sigma \vdash_{\emptyset}^{\emptyset} \Delta [ok]}{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} \forall_{\sigma}^+(v) : \forall a : \sigma. T} \text{ (ty-}\forall\text{-intro)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} e : \forall a : \sigma. T \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} \forall_{\sigma}^-(e) : T[a \mapsto s]} \text{ (ty-}\forall\text{-elim)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} v : T[a \mapsto s] \quad \Sigma \vdash s : \sigma}{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} \exists_{\sigma}(v) : \exists a : \sigma. T} \text{ (ty-}\exists\text{-intro)} \\
 \\
 \frac{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} e_1 : \exists a : \sigma. T_1 \quad \Sigma, a : \sigma; \Delta, x : T_1 \vdash_{\emptyset}^{\emptyset} e_2 : T_2}{\Sigma; \Delta \vdash_{\emptyset}^{\emptyset} \mathbf{let } \exists_{\sigma}(x) = e_1 \mathbf{ in } e_2 \mathbf{ end} : T_2} \text{ (ty-}\exists\text{-elim)}
 \end{array}$$

 Figure 7: The typing rules for λ_{dist}^+

The extension from λ_{dist} to λ_{dist}^+ corresponds tightly to the one from λ_{code} to λ_{code}^+ (Chen and Xi 2003), but a key difference lies in the treatment of *Lift*. The rather technical presentation of the rest of this section essentially shows that λ_{dist}^+ can be assigned a static semantics that is justified by a translation from λ_{dist}^+ to λ_{dist} . By assuming this, the reader may simply skip the rest of this section.

The dynamic variable context Δ is now defined as follows,

$$\text{dyn. ctx. } \Delta ::= \emptyset \mid \Delta, xf@k : T$$

where $xf@k$ are called staged variables at level $k \geq 0$. Intuitively, an expression e in the empty evaluation context is said to be at level 0; if an occurrence of e in e_0 is at level k , then the occurrence of e in \hat{e}_0 is at level $k + 1$; if an occurrence of e in e_0 is at level $k + 1$, then the occurrence of e in \hat{e}_0 is at level k ; if an occurrence of **lam** $x. e_1$ or **fix** $f. e_1$ is at level k , then x or f is bound at level k . A declared staged variable $xf@k$ in Δ simply indicates that xf is to be bound at level k .

3.1 Static Semantics

We define environment mappings \mathcal{G} and location mappings \mathcal{L} as follows:

$$\begin{aligned} \text{env. mappings } \mathcal{G} & ::= \emptyset \mid \mathcal{G} + G \\ \text{loc. mappings } \mathcal{L} & ::= \emptyset \mid \mathcal{L} + L \end{aligned}$$

The length of environment mappings \mathcal{G} is defined as follows:

$$|\emptyset| = 0 \quad |\mathcal{G} + G| = |\mathcal{G}| + 1$$

Given $1 \leq k \leq |\mathcal{G}|$ for some environment mapping $\mathcal{G} = \mathcal{G}' + G$, $\mathcal{G}(k) = G$ if $k = |\mathcal{G}|$, or $\mathcal{G}(k) = \mathcal{G}'(k)$ if $k \leq |\mathcal{G}'|$. Similar notations also apply to location mappings \mathcal{L} . In the following presentation, we use \mathcal{G}/\mathcal{L} for an environment mapping \mathcal{G} and a location mapping \mathcal{L} of the same length. We write $\Sigma \vdash_{\mathcal{L}}^{\mathcal{G}} \Delta [ok]$ to mean that

1. $\Sigma \vdash \Delta(xf) : \text{type}$ for each $xf \in \mathbf{dom}(\Delta)$, and
2. $|\mathcal{G}| = |\mathcal{L}|$, and
3. $\Delta \vdash \mathcal{G}(k) : \text{env}$ for each $1 \leq k \leq |\mathcal{G}|$, and
4. $\Delta \vdash \mathcal{L}(k) : \text{loc}$ for each $1 \leq k \leq |\mathcal{L}|$.

In addition, we introduce the following definitions:

- Given $G, k > 0$ and Δ , we define $G(k; \Delta)$ as follows:

$$\begin{aligned} G(k; \emptyset) & = G & ; \\ G(k; \Delta, xf@k' : T) & = T :: G(k; \Delta) & \text{ if } k = k'; \\ G(k; \Delta, xf@k' : T) & = G(k; \Delta) & \text{ if } k \neq k'. \end{aligned}$$

- Given $\mathcal{G}/\mathcal{L}, \Delta$ and T , we define

$$\mathcal{G}/\mathcal{L}(0; \Delta; T) = T,$$

and for $1 \leq k \leq |\mathcal{G}| = |\mathcal{L}|$,

$$\mathcal{G}/\mathcal{L}(k; \Delta; T) = \mathcal{G}/\mathcal{L}(k-1; \Delta; \langle L, G(k; \Delta), T \rangle),$$

where $L = \mathcal{L}(k)$ and $G = \mathcal{G}(k)$.

We write $\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e : T$ for a typing judgment in λ_{dist}^+ , where we require $\Sigma \vdash_{\mathcal{L}}^{\mathcal{G}} \Delta [ok]$. Intuitively, for each $1 \leq k \leq |\mathcal{G}|$, $\mathcal{G}(k)$ and $\mathcal{L}(k)$ stand for the type environment and location for code at level k . We present the typing rules for λ_{dist}^+ in Figure 7. Note that expressions of quantified types are only allowed at level 0. This restriction is to be removed in Section 4.

Assume $n \geq 1$

$$\begin{aligned} \text{exec}(\text{Lift}_{n+1}(e)) &= \text{Lift}_n(\text{dec}(e)) \\ \text{exec}(\text{One}_{n+1}) &= \text{One}_n \\ \text{exec}(\text{Shi}_{n+1}(e)) &= \text{Shi}_n(\text{exec}(e)) \\ \text{exec}(\text{Lam}_{n+1}(e)) &= \text{Lam}_n(\text{exec}(e)) \\ \text{exec}(\text{App}_{n+1}(e_1, e_2)) &= \text{App}_n(\text{exec}(e_1), \text{exec}(e_2)) \\ \text{exec}(\text{Fix}_{n+1}(e)) &= \text{Fix}_n(\text{exec}(e)) \end{aligned}$$

Figure 8: Extending the function *exec*

3.2 A Translation from λ_{dist}^+ to λ_{dist}

We now introduce generalized typeful code constructors $\text{Lift}_n, \text{One}_n, \text{Shi}_n, \text{Lam}_n, \text{App}_n$ and Fix_n in Figure 6, which are needed for representing expressions in λ_{dist}^+ at level $n \geq 1$. These constructors can already be defined in terms of the constructors $\text{Lift}, \text{One}, \text{Shi}, \text{Lam}, \text{App}$ and Fix . However, the definition is rather involved and we refer the interested reader to (Chen and Xi 2003; Chen and Xi 2004). In Figure 8, we extend the function *exec* to deal with the generalized code constructors.

We now use $\overline{xf}s$ for a sequence of staged variables, that is, $\overline{xf}s$ is of the form $xf_1 @ k_1, \dots, xf_n @ k_n$. For each $k > 0$, we define $\text{var}_k(\overline{xf}s; xf)$ as follows under the assumption that $xf @ k$ occurs in $\overline{xf}s$: for $\overline{xf}s = (\overline{xf}s_1, xf_1 @ k_1)$, $\text{var}_k(\overline{xf}s; xf)$ is

$$\begin{aligned} \text{One}_k & \text{ if } k_1 = k \text{ and } xf_1 = xf; \\ \text{Shi}_k(\text{var}_k(\overline{xf}s_1; xf)) & \text{ if } k_1 = k \text{ and } xf_1 \neq xf; \\ \text{var}_k(\overline{xf}s_1; xf) & \text{ if } k_1 \neq k \end{aligned}$$

Also, we use $\text{App}_k^n(e_0, e_1, \dots, e_n)$ for e_0 if $n = 0$, or for $\text{App}_k(\text{App}_k^{n-1}(e_0, \dots, e_{n-1}), e_n)$ if $n > 0$.

In Figure 9, we define a translation $\text{trans}_k(\cdot; \cdot)$ for each $k \geq 0$ that translates expressions in λ_{dist}^+ into those in λ_{dist} . A crucial property of $\text{trans}_k(\cdot; \cdot)$ is captured by the following lemma.

$trans_0(\cdot; \cdot)$

$$\begin{aligned}
 trans_0(\overline{xf}; xf) &= xf \quad \text{if } xf@0 \text{ occurs in } \overline{xf} \\
 trans_0(\overline{xf}; c(e_1, \dots, e_n)) &= c(trans_0(\overline{xf}; e_1), \dots, trans_0(\overline{xf}; e_n)) \\
 trans_0(\overline{xf}; \mathbf{lam} x. e) &= \mathbf{lam} x. trans_0(\overline{xf}, x@0; e) \\
 trans_0(\overline{xf}; e_1(e_2)) &= trans_0(\overline{xf}; e_1)(trans_0(\overline{xf}; e_2)) \\
 trans_0(\overline{xf}; \mathbf{fix} f. e) &= \mathbf{fix} f. trans_0(\overline{xf}, f@0; e) \\
 trans_0(\overline{xf}; \forall_\sigma^+(e)) &= \forall_\sigma^+(trans_0(\overline{xf}; e)) \\
 trans_0(\overline{xf}; \forall_\sigma^-(e)) &= \forall_\sigma^-(trans_0(\overline{xf}; e)) \\
 trans_0(\overline{xf}; \exists_\sigma(e)) &= \exists_\sigma(trans_0(\overline{xf}; e)) \\
 trans_0(\overline{xf}; \mathbf{let} \exists_\sigma(x) = e_1 \mathbf{in} e_2 \mathbf{end}) &= \mathbf{let} \exists_\sigma(x) = trans_0(\overline{xf}; e_1) \mathbf{in} trans_0(\overline{xf}, x; e_2) \mathbf{end} \\
 trans_0(\overline{xf}; \cdot(e)) &= trans_1(\overline{xf}; e)
 \end{aligned}$$

$trans_k(\cdot; \cdot)$ for $k \geq 1$

$$\begin{aligned}
 trans_k(\overline{xf}; xf) &= var_k(\overline{xf}; xf) \quad \text{if } xf@k \text{ occurs in } \overline{xf} \\
 trans_k(\overline{xf}; c(e_1, \dots, e_n)) &= App_k^n(Lift_k(\$^k c), trans_k(\overline{xf}; e_1), \dots, trans_k(\overline{xf}; e_n)) \\
 trans_k(\overline{xf}; \mathbf{lam} x. e) &= Lam_k(trans_k(\overline{xf}, x@k; e)) \\
 trans_k(\overline{xf}; e_1(e_2)) &= App_k(trans_k(\overline{xf}; e_1), trans_k(\overline{xf}; e_2)) \\
 trans_k(\overline{xf}; \mathbf{fix} f. e) &= Fix_k(trans_k(\overline{xf}, f@k; e)) \\
 trans_k(\overline{xf}; \cdot(e)) &= trans_{k+1}(\overline{xf}; e) \\
 trans_k(\overline{xf}; \hat{\cdot}(e)) &= trans_{k-1}(\overline{xf}; e)
 \end{aligned}$$

Figure 9: The extended definition of $trans_k(\cdot; \cdot)$ for $k \geq 0$

Lemma 3.1 Assume that $\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e : T$ is derivable in λ_{dist}^+ and Δ is of the form $xf_1@k_1 : T_1, \dots, xf_n@k_n : T_n$. Then, $\Sigma; (\Delta)_0 \vdash trans_{|\mathcal{L}|}(xf_1, \dots, xf_n; e) : \mathcal{G}/\mathcal{L}(\Delta; T)$ is derivable in λ_{dist} , where $(\Delta)_0$ is defined as follows:

$$\begin{aligned} (\emptyset)_0 &= \emptyset && ; \\ (\Delta, x@0 : T) &= (\Delta)_0, x : T && ; \\ (\Delta, x@k : T) &= (\Delta)_0 && \text{if } k > 0. \end{aligned}$$

Proof The proof follows from structural induction on the typing derivation of $\Sigma; \Delta \vdash_{\mathcal{L}}^{\mathcal{G}} e : T$. ■

Given an expression e in λ_{dist}^+ , we write $trans(e)$ for $trans_0(\emptyset; e)$ (if it is well-defined) and call it the translation of e .

Theorem 3.2 Assume that $\emptyset; \emptyset \vdash_{\emptyset}^{\emptyset} e : T$ is derivable. Then $\emptyset; \emptyset \vdash trans(e) : T$ is derivable.

Proof This immediately follows from Lemma 3.1. ■

The programmer can now construct a distributed program in λ_{dist}^+ that may (and probably should) make use of meta-programming syntax and then assign it the dynamic semantics of its translation in λ_{dist} . In other words, the meta-programming syntax can be treated as a form of syntactic sugar. This is precisely the significance of Theorem 3.2.

$$\begin{aligned} Uni_{\sigma}^+ &: \forall \lambda. \forall \gamma. \forall a_1 : \sigma \rightarrow type. (\forall a_2 : \sigma. \langle \lambda, \gamma, a_1(a_2) \rangle) \Rightarrow \langle \lambda, \gamma, \forall a : \sigma. a_1(a) \rangle \\ Uni_{\sigma}^- &: \forall \lambda. \forall \gamma. \forall a_1 : \sigma \rightarrow type. \forall a_2 : \sigma. (\langle \lambda, \gamma, \forall a : \sigma. a_1(a) \rangle) \Rightarrow \langle \lambda, \gamma, a_1(a_2) \rangle \\ Exi_{\sigma}^+ &: \forall \lambda. \forall \gamma. \forall a_1 : \sigma \rightarrow type. \forall a_2 : \sigma. (\langle \lambda, \gamma, a_1(a_2) \rangle) \Rightarrow \langle \lambda, \gamma, \exists a : \sigma. a_1(a) \rangle \\ Exi_{\sigma}^- &: \forall \lambda. \forall \gamma. \forall a_1 : \sigma \rightarrow type. \forall \alpha. (\langle \lambda, \gamma, \exists a : \sigma. a_1(a) \rangle, \forall a : \sigma. \langle \lambda, a_1(a) :: \gamma, \alpha \rangle) \Rightarrow \langle \lambda, \gamma, \alpha \rangle \end{aligned}$$

Figure 10: The typeful code constructors for quantified types

4 Extensions

It is straightforward to extend λ_{dist} (and subsequently λ_{dist}^+) to support additional language features such as conditionals, pairs, sums, references, etc. Please refer to (Chen and Xi 2003) for further details. What is interesting is to support typeful code constructors that can represent code of either universally or existentially quantified types. The need for such code constructors is to be demonstrated convincingly in an example presented later in the paper.

The code constructors $Uni_{\sigma}^+, Uni_{\sigma}^-, Exi_{\sigma}^+, Exi_{\sigma}^-$ and the c-types assigned to them are given in Figure 10. Please note that we need to extend λ_{dist} with second-order sort quantification in order to accommodate these c-types. Unsurprisingly, the function *eval* outlined on page 3 needs to be extended as follows to handle the new code constructors:

```
fun eval (p, env) = (* 'env' is a list of values *)
  case p of
```

```

...
| UniPlus p = eval (p, env)
| UniMinus p = eval (p, env)
| ExiPlus p = eval (p, env)
| ExiMinus (p1, p2) = eval (p2, eval (p1, env) :: env)
    
```

where the meaning of *UniPlus* *UniMinus* *ExiPlus* and *ExiMinus* should be obvious. We may have to check at run-time whether p represents a value when evaluating $UniPlus(p)$ if the form of value restriction in λ_{dist} also needs to be imposed at code level. We present in Figure 11 the generalized typeful code constructors corresponding to Uni_{σ}^{+} , Uni_{σ}^{-} , Exi_{σ}^{+} , Exi_{σ}^{-} , and then extend the definition of the translation functions $trans_k(\cdot; \cdot)$ for $k \geq 1$ in Figure 12. We omit the details on extending the function *exec* to handle these generalized typeful code constructors, which is straightforward.

$$\begin{aligned}
 Uni_{\sigma,n}^{+} & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall a_1 : \sigma \rightarrow type. \\
 & \quad (\forall a_2 : \sigma. \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, a_1(a_2) \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \forall a : \sigma. a_1(a) \rangle \dots \rangle \rangle \\
 Uni_{\sigma,n}^{-} & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall a_1 : \sigma \rightarrow type. \forall a_2 : \sigma. \\
 & \quad (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \forall a : \sigma. a_1(a) \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, a_1(a_2) \rangle \dots \rangle \rangle \\
 Exi_{\sigma,n}^{+} & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall a_1 : \sigma \rightarrow type. \forall a_2 : \sigma. \\
 & \quad (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, a_1(a_2) \rangle \dots \rangle \rangle) \Rightarrow \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \exists a : \sigma. a_1(a) \rangle \dots \rangle \rangle \\
 Exi_{\sigma,n}^{-} & : \forall \lambda_1 \dots \forall \lambda_n. \forall \gamma_1 \dots \forall \gamma_n. \forall a_1 : \sigma \rightarrow type. \forall a_2 : \sigma. \\
 & \quad (\langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, \exists a : \sigma. a_1(a) \rangle \dots \rangle \rangle, \forall a : \sigma. \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, a_1(a) :: \gamma_n, a_2 \rangle \dots \rangle \rangle) \Rightarrow \\
 & \quad \langle \lambda_1, \gamma_1, \langle \dots \langle \lambda_n, \gamma_n, a_2 \rangle \dots \rangle \rangle
 \end{aligned}$$

Figure 11: The additional generalized typeful code constructors

$$\begin{aligned}
 trans_k(\overline{xf}s; \forall_{\sigma}^{+}(e)) & = Uni_{\sigma,k}^{+}(trans_k(\overline{xf}s; e)) \\
 trans_k(\overline{xf}s; \forall_{\sigma}^{-}(e)) & = Uni_{\sigma,k}^{-}(trans_k(\overline{xf}s; e)) \\
 trans_k(\overline{xf}s; \exists_{\sigma}(e)) & = Exi_{\sigma,k}^{+}(trans_k(\overline{xf}s; e)) \\
 trans_k(\overline{xf}s; \text{let } \exists_{\sigma}(x) = e_1 \text{ in } e_2 \text{ end}) & = Exi_{\sigma,k}^{-}(trans_k(\overline{xf}s; e_1), trans_k(\overline{xf}s, x@k; e_2))
 \end{aligned}$$

 Figure 12: The extended definition of $trans_k(\cdot; \cdot)$ for $k \geq 1$

5 Examples of Distributed Meta-Programs

We now need an external language for the programmer to construct distributed programs and then a process to translate such programs into typing derivations in (properly extended) λ_{dist}^{+} . The following is a brief description of the syntax for this external languages.

```

(* 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

(* '% (...)' is a shorthand for ^(Lift (...)) *)
(* 'encInt' encodes integers into messages *)
(* '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

(* 'code (L, G, T)' is for the code type <L, G, T> *)
(* 'nil' for empty typing environment *)
(* 'fix f x => ...' means 'fix f. lam x => ...' *)

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

```

Figure 13: RPC vs. Mobile Code

$$\begin{aligned}
 \text{expressions } e ::= & x \mid f \mid c(e_1, \dots, e_n) \mid \mathbf{if}(e_1, e_2, e_3) \mid \\
 & \mathbf{lam } x. e \mid \mathbf{lam } x : T. e \mid e_1(e_2) \mid \\
 & \mathbf{fix } f. e \mid \mathbf{fix } f[a_1 : \sigma_1, \dots, a_n : \sigma_n] : T. e \mid \\
 & \mathbf{let } x = e_1 \mathbf{in } e_2 \mathbf{end} \mid (e : T) \mid \\
 & \text{'}(e) \mid \text{^}(e) \mid \text{\%}(e)
 \end{aligned}$$

The only syntax that may seem unfamiliar is

$$\mathbf{fix } f[a_1 : \sigma_1, \dots, a_n : \sigma_n] : T. e,$$

which indicates that the fixed-point expression is expected to be assigned the type $\forall a_1 : \sigma_1 \dots \forall a_n : \sigma. T$. Also, we use $\text{\%}(\cdot)$ as a shorthand for $\text{^}(Lift(\cdot))$.

Of course, we are also in need of an approach that can effectively elaborate programs in this external language into λ_{dist}^+ . While we cannot formally describe such an approach at this moment, we reasonably expect that the reader can readily relate the programming examples presented in this section to their corresponding parts in λ_{dist}^+ .

RPC vs. Mobile Code We present an example in Figure 13 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 call 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.
- *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.

Recursive Code Propagation In Figure 14, we present an implementation of the Fibonacci function in which a programming method called *recursive code propagation* is involved. Though short, this example may seem difficult to understand, and we provide some explanation as follows.

We use *fib*(*n*) for the *n*th Fibonacci number, where *n* ranges over natural numbers. Assume the existence of a function *choose* of c-type $() \Rightarrow \exists \lambda. \mathbf{loc}(\lambda)$ at every location, which can be called to generate locations. The value *fibCode* represents some code for computing Fibonacci numbers in the following manner: Given a natural number *n*, if $n \leq 1$ then 1 is returned; otherwise, the function *choose* is called twice to select two locations *L*₁ and *L*₂ to which the code represented by *fibCode* is sent for computing *fib*(*n* − 1) and *fib*(*n* − 2), respectively, and the results are fetched back and then added up.

It is a routine verification that *fibCode* can be assigned the type *FibCodeType*, which is defined to be the following recursive type:

$$\mu \alpha. \forall \lambda. \langle \lambda, \epsilon, \alpha \rightarrow \mathbf{int} \rightarrow \mathbf{int} \rangle$$

```
typedef rec FibCodeType = (* recursive type def. *)
  {l:loc} code (l, nil, FibCodeType -> int -> int)

(* 'encCode' encodes a value of type 'FibCodeType' *)

val fibCode: FibCodeType =
  `(lam f => lam n =>
    if n <= 1 then 1
    else
      let
        val L1 = choose () (* select a location *)
        val L2 = choose () (* select a location *)
      in
        rexecInt
          (L1, `(^f %(encCode f) %(encInt(n-1)))) +
        rexecInt
          (L2, `(^f %(encCode f) %(encInt(n-2))))
      end)

val fib: int -> int = exec fibCode fibCode
```

Figure 14: Computing Fibonacci Numbers

The value *fibCode* represents some closed code that can be executed at any location L to obtain a function of the type $FibCodeType \rightarrow \mathbf{int} \rightarrow \mathbf{int}$; this function, when applied to *fibCode*, returns a function of type $\mathbf{int} \rightarrow \mathbf{int}$ that computes Fibonacci numbers.

To further demonstrate the potential use of recursive code propagation, we outline an implementation of peer-to-peer file search function in Figure 15. We assume that a function *searchFile* is available at every location that can be called to search a file. This implementation essentially uses the same strategy as the one in Figure 14 to propagate code recursively. Given the name of a file and an integer (which determines the maximum number of times that the code for search is allowed to propagate), the function *p2pSearch* creates a reference *result* for storing the search results; then it forms a function *store* and a name for *store* to allow *result* to be updated remotely; then the code for search is constructed and then executed; the code checks if the file being searched can be found locally; if so, a link to the file is written into *result*; otherwise, two more locations are generated (if the code for search is still allowed to propagate) and the code for search is sent to these locations. Note that this example involves pattern matching at code level, for which some detailed explanation can be found in (Chen and Xi 2004).

6 Related Work

There have been some recent studies that advocate the use of modal logic in designing type systems to support distributed programming (Moody 2003; Jia and Walker 2003; Murphy, Crary, Harper, and Pfenning 2004), 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.

A type system based on the modal logic S4 is developed in (Moody 2003) to support typed distributed programming, where the connection relation between worlds (which we call locations) is assumed to be reflexive and transitive but not necessarily symmetric and is directly dealt with in the type system. This is a design with a focus on supporting grid computing and it may not be flexible enough to handle applications where connections between worlds may change from time to time. Also, while a program in this system can be assigned a type to indicate the mobility of the program, program transmission from one location to another is completely hidden from the programmer. On the one hand, by relying on the run-time system to fully take care of data/code distribution on behalf of the programmer, this design has the advantage of providing the programmer with a simpler type system and thus possibly a cleaner programming interface. On the other hand, the programmer is left with little control over issues such as determining the actual locations for executing mobile code and/or providing services.

Another type system for supporting distributed computing is introduced in (Murphy, Crary, Harper, and Pfenning 2004), which bears some close resemblance to (Moody 2003). This type system, with its basis in the modal logic S5, assumes that all worlds are interconnected and thus has no need for explicitly dealing with world accessibility. Also, λ_{rpc} -calculus (Jia and Walker 2003)

```

typedef rec P2PCodeType = (* recursive type def. *)
  {l:loc} code (l, nil, P2PCodeType -> int -> unit)

(* 'encFilename' encodes a filename *)
(* 'encCode' encodes a value of type 'P2PCodeType' *)
(* 'encLoc' encodes a location *)

fun p2pCode (filename: filename,
            store: (url -> unit) @ here): P2PCodeType =
  `(lam f => lam n =>
    if n = 0 then ()
    else
      case searchFile (encFilename filename) of
      | NONE =>
        (* propagate code to two neighbors, where *)
        (* the number 2 is chosen arbitrarily *)
        let
          val L1 = choose ()
          val L2 = choose ()
        in
          rexecUnit
            (L1, `(^f encCode f encInt(n-1)));
          rexecUnit
            (L2, `(^f encCode f encInt(n-1)))
        end
      | SOME url => (* use rpc to store the url *)
        (* 'rget' is for the type 'url -> unit' *)
        rget (encLoc here), (n2m store)) (url)
    end)

(*
 * 'level' determines the maximum number of time code
 * propagations can be performed.
 *)

fun p2pSearch (filename: filename, level: int): unit =
  let
    (* search results are to be stored in 'result' *)
    val result: url list = ref []

    (* 'store' is to be called to store a url *)
    fun store (x: url) = (result := x :: !result)

    val code: P2PCodeType = p2pCode(filename, put store)
  in
    exec code code level
  end
end

```

Figure 15: Peer-to-Peer File Search

is developed recently to support typed distributed programming, which rests upon an extension of the modal logic S5 with some hybrid-logic features. In both (Murphy, Crary, Harper, and Pfenning 2004) and (Jia and Walker 2003), locations can occur in a program to indicate where certain parts of the program should be distributed at compile-time. However, locations are still not treated as first-class values. As a consequence, it seems that recursive code propagation as is implemented in Figure 14 and Figure 15, which makes use of run-time generated locations, cannot be handled.

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 (Moody 2003) or explicit (Murphy, Crary, Harper, and Pfenning 2004; Jia and Walker 2003). 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} . With this addition, the essential features in the above systems can then be readily handled in λ_{dist} . In particular, the modality operators \Box and \Diamond can be encoded as follows in λ_{dist} :

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

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, and we present some examples as follows in support of this claim.

- A proof term for $\Box T \rightarrow T$ is $\mathbf{lam} x. \mathit{exec}(x)$, which means that a value of type $\Box T$ can be executed locally to obtain a value of type T .
- A proof term for $T \rightarrow \Diamond T$ is $\mathbf{lam} x. \exists(\mathit{here}, \mathit{put}(x))$, which indicates that given a value v of type T , we can call put on v to generate a name for v and then pair it with the location here to form a value of type $\Diamond T$.
- A proof term for $(\Diamond T_1 \rightarrow \Box T_2) \rightarrow \Box(T_1 \rightarrow T_2)$ is the following function written in the syntax of λ_{dist}^+ :

$$\mathbf{lam} f. \langle \mathbf{lam} x. \mathit{exec}(\mathit{rexec}_{\Box T_2}(\%(\mathit{enc}(\mathit{here})), \langle \%(\mathit{enc}(n2m(\mathit{put}(f)))) \%(\mathit{enc}(\exists(\mathit{here}, \mathit{put}(x)))) \rangle \rangle)) \rangle$$

The meaning of the proof term is somewhat involved. When applied to a function f of type $\Diamond T_1 \rightarrow \Box T_2$ at location L_0 , the above proof term yields some code that can be transmitted to another location L_1 and then be executed there to generate a function f' such that f' , when applied to a value v , forms a pair consisting of L_1 and a name for v , and then sends the pair to L_0 to compute the result of applying f to the pair, and then fetches back the result, which represents some closed code, and then executes the result.

Theoretical models of distributed/concurrent computation are often built upon process algebras, some of which can be found in (Berry and Boudol 1992; Milner, Parrow, and Walker 1992; Cardelli and Gordon 2000; Fournet, Gonthier, Lévy, Maranget, and Rémy 1996; Schmitt and Stefani 2003; Yoshida and Hennessy 1999). 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. In a loose sense, we are probably more concerned with the programming issue of distributed computing, while these works based on process algebras have a stronger focus on the semantics of distributed computing.

We have already stated in the introduction that the typeful code representation in this paper is largely adopted from a previous study on meta-programming (Chen and Xi 2003), which in turn is closely related to the notion of guarded recursive datatypes (Xi, Chen, and Chen 2003). In the language λ_{code} (Chen and Xi 2003), the type for code is of the form $\langle G, T \rangle$, and this form is modified to $\langle L, G, T \rangle$ to include the location of code. This seemingly minor modification, however, leads to many rather significant changes in the design and formalization of λ_{dist} , some of which are mentioned as follows:

- The set of special functions in Figure 3 are chosen to support data being moved between locations and code being executed remotely, and the abstract type constructors $@$ and \mathbf{msg} are introduced so as to assign proper types to these special functions.
- The dynamic semantics of λ_{dist} needs to deal with the notion of values stored at remote locations, which simply does not exist in λ_{code} .
- In order to guarantee the mobility of values that represent code, the constructor *Lift*, which is allowed to be applied to an arbitrary value to form code in λ_{code} , can only be applied to messages to form code in λ_{dist} . In particular, λ_{code} cannot be simply considered a special instance of λ_{dist} because of the different manners in which *Lift* is handled in λ_{code} and λ_{dist} .
- The need for constant messages like those in Figure 4 is not shared by λ_{code} , in which the distinction between values and the messages representing these values simply does not exist.

When programming is concerned, the use of recursive types to support recursive code propagation as is presented in Figure 14 and Figure 15 is entirely novel and nontrivial. Moreover, λ_{dist} is extended with the code constructors Uni^+ , Uni^- , Exi^+ and Exi^- for forming code of universally and existentially quantified types, which are not studied in λ_{code} . For those who are familiar with the framework *Applied Type System (ATS)*, we point out that λ_{code} can be classified as an applied type system but λ_{dist} cannot as the notion of remotely located values in dynamic semantics of the latter does not exist in *ATS*.

7 Conclusion

We have presented a simple and general approach to support mobile computing through distributed meta-programming that allows code to be generated at run-time and then sent to a remote location for execution. To guarantee that code generated at run-time is well-typed and can only be executed

at locations where adequate resources exist, we make use of a form of type code representation developed in a previous study on meta-programming (Chen and Xi 2003; Chen and Xi 2004). The main contribution of the paper lies in the recognition and then formalization of this novel approach that combines, for the first time, 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

- Berry, G. and G. Boudol (1992). The chemical abstract machine. *Theoretical Computer Science* 96(1), 217–248.
- Cardelli, L. and A. D. Gordon (2000, June). Mobile Ambients. *Theoretical Computer Science* 240(1), 177–213. Special Issue on Coordination edited by D. Le Mtayer.
- Chen, C. and H. Xi (2003, August). Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, pp. 169–180.
- Chen, C. and H. Xi (2004, February). Meta-Programming through Typeful Code Representation. Available at:
www.cs.bu.edu/~hwx/academic/drafts/MPTCR.ps.
- Church, A. (1940). A formulation of the simple type theory of types. *Journal of Symbolic Logic* 5, 56–68.
- de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies. *Indagationes mathematicae* 34, 381–392.
- Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy (1996). A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, pp. 406–421. Springer-Verlag LNCS 1119.
- Jia, L. and D. Walker (2003). Modal Proofs as Distributed Programs. Technical Report TR-671-03, Princeton University.
- Milner, R., J. Parrow, and D. Walker (1992). A calculus of processes, parts I and II. *Information and Computation* 100, 1–40 and 41–77.
- Moody, J. (2003). Modal Logic as a Basis for Distributed Computation. Technical Report CMU-CS-03-194, Carnegie Mellon University.
- Murphy, T., K. Crary, R. Harper, and F. Pfenning (2004). A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of 19th IEEE Symposium on Logic in Computer Science (LICS)*. (to appear).
- Pfenning, F. *Computation and Deduction*. Cambridge University Press. (to appear).
- Pfenning, F. and C. Elliott (1988, June). Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, Atlanta, Georgia, pp. 199–208.

- Schmitt, A. and J.-B. Stefani (2003, January). The M-calculus: a Higher-Order Distributed Process Calculus. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, New Orleans, LA, pp. 50–61.
- Taha, W. and T. Sheard (2000). MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248(1-2), 211–242.
- Xi, H. (2004). Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pp. 394–408. Springer-Verlag LNCS 3085.
- Xi, H., C. Chen, and G. Chen (2003, January). Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, New Orleans, LA, pp. 224–235.
- Yoshida, N. and M. Hennessy (1999). Subtyping and Locality in Distributed Higher-Order Processes. In *Proceedings of CONCUR'99*, pp. 557–573. Springer-Verlag LNCS 1664.