

# MobileML : A Programming Language for Mobile Computation

Masatomo Hashimoto and Akinori Yonezawa

Department of Information Science, Faculty of Science, University of Tokyo,  
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan  
{masatomo,yonezawa}@is.s.u-tokyo.ac.jp

**Abstract.** This paper describes a language which facilitates programming for *mobile computation* i.e., computation in which code migrates and continues to run across network nodes. Such languages allow us to develop novel distributed applications (such as workflow systems, flexible software distribution, and intelligent search systems) more easily and efficiently. However, many of existing programming language systems are often insufficient because they lack the support for concise description of migration and formal models for reasoning about program's properties including type safety and security. Our goal is to construct a programming language system which adequately supports mobile computation on a solid theoretical basis. As an attempt to achieve that, we have designed a programming language based on ML which has well-founded theoretical bases. The features of this language include: transparent migration, dynamic linking with distributed resources by means of *contexts*, and semantics consistent with the original ML. Especially, our notion of contexts allows us to succinctly describe the interaction between mobile code and environments at destination nodes. We briefly explain a simple semantic model based on Plotkin's  $\lambda_v$ -calculus and *tuple spaces*. Though our model currently guarantees only type soundness, we believe that theoretical results such as prevention of security violation (of Heintze and Rieckes' SLam Calculus) can be incorporated into our system rather easily. We have also implemented an experimental interpreter system as a first step of the full-fledged language system.

## 1 Introduction

With explosive growth of global computer networks, programming languages are expected to have some sophisticated mechanisms for networkwide programming. Java [12] popularized the notion of *mobile code* which can travel on a heterogeneous network and automatically runs upon arrival at the destination. Some of modern distributed languages [4, 13, 28, 20, 25] facilitate a novel form of distributed computation, called *mobile computation* [28, 13, 5] i.e., computation in which code migrates and continues to run across network nodes. Mobile computation is expected to be a promising framework for construction of distributed systems. The main advantages of constructing a distributed system in the style of mobile computation are efficiency and flexibility. For example, suppose that

we are trying to search the Internet for routes of a tour and to reserve tickets from a mobile terminal. Connection between mobile terminals and the Internet is usually very poor in speed and too expensive in cost. Therefore, we would like to avoid connecting to the Internet for a long period. In the framework of mobile computation, search and reservation programs migrate from mobile terminals to servers of travel agencies on the Internet [15, 28].

It is desirable for programming languages for mobile computation to have at least the following features:

*Transparent migration.* Mobile code is a program which migrates from one node to another in a network. It can suspend its execution at arbitrary program points, move to another node, and resume execution with exactly the same execution state at the destination node. We call such a form of migration *strong* or *transparent* [13]. Transparent migration is desirable because it allows succinct description of migration and offers clear semantics of migration [13]. This feature reduces programmer's burden considerably.

*Abstraction of local environments.* When a piece of code moves to a network node, its execution environment also changes. In the style of mobile computation, pieces of mobile code naturally make use of local environments including processors, displays, hard disks, and local data such as strings, images and sounds. Therefore, appropriate abstractions for local environments are necessary in order to simplify programming.

*Formal models.* In mobile systems, external code may be executed at a user's machine and user's private code may be executed elsewhere in the network. Therefore security and safety are important issues. Solid formal models are the key to formal reasoning about such properties of programs.

However, none of the existing programming languages satisfy all the above. Obliq [4] is a distributed object-oriented language based on static scoping rules. In Obliq, we can transmit closures to other network nodes, and construct mobile applications [3]. However, transparent migration and abstractions for local environments are not supported. Telescript [28] is a pioneer of mobile language. It can describe migration by only writing "**go destination**" at almost an arbitrary program point. Although Telescript supports transparent migration, it does not have the notion of local environment and any formal model. D'Agents [14], once called AgentTcl [13], is based on a scripting language Tcl and supports transparent migration. However, no formal model exists. JavaGo [25] realizes transparent migration by source to source transformation in Java with Remote Method Invocation. Its transformation mechanism is based on a formal framework [26]. However, Java language itself is far from rigorously specified. Distributed Join-Calculus [10] is a distributed concurrent language based on a solid formal model. This language can represent *locations* which contain running processes, and may fail. However, it cannot directly describe utilization of local environments.

Our goal is to construct a programming language system which adequately supports mobile computation on a rigorous theoretical basis. As an approach to that goal, we have designed a language called *MobileML* based on ML [6], and implemented an experimental interpreter system. The main features of this language include: transparent migration, dynamic linking with distributed resources by means of *contexts*, and semantics consistent with the original ML. Above all, our notion of contexts allows us to succinctly describe various forms of interactions between mobile code and environments at the destination nodes. (Examples include a description of runtime update of libraries used in an application.) Of course, we can utilize theoretical results and implementation techniques developed for the family of ML languages such as OCaml and SML/NJ. We have also developed a simple semantic model for MobileML based on Plotkin’s  $\lambda_v$ -calculus and *tuple spaces* [11].

The rest of this paper is organized as follows. Section 2 introduces the key notions in our system. Then MobileML system is described in Sect. 3. Section 4 outlines a semantic model for MobileML. After coding examples are given in Sect. 5, Sect. 6 mentions a related work, and finally Sect. 7 concludes this paper. A summary of a formal model for MobileML is presented in the Appendix.

## 2 Key Notions

### 2.1 Transparent Migration

When a computation transparently migrates, it suspends its execution, moves to another machine, and resumes its execution at the destination machine preserving all of the execution states. Suppose that we are evaluating the following expression, assuming the standard call-by-value semantics.

$$\mathbf{let } f = \lambda x.x + 1 \mathbf{ in } ( \mathbf{print\_int } ( f \ 3 ) )$$

The evaluation can be divided into two phases:

- (1) evaluation of  $\lambda x.x + 1$  (and binding  $f$  to the result), and
- (2) evaluation of  $( \mathbf{print\_int } ( f \ 3 ) )$ .

If we want the second phase to be executed somewhere else, say at a destination  $l$ , all we have to do is to put a **go** expression as follows:

$$\mathbf{let } f = \lambda x.x + 1 \mathbf{ in } ( \mathbf{go } l; \mathbf{print\_int } ( f \ 3 ) )$$

Evaluating the above expression makes the code of “ $\mathbf{print\_int } ( f \ 3 )$ ” be sent to and executed at the destination designated by  $l$ . Note that the binding for  $f$  is kept throughout the migration, and then  $(\lambda x.x + 1) \ 3$  is executed at the destination. The state of the execution stack is also preserved. Consider the expression below.

$$\begin{aligned} &\mathbf{let } f = \lambda x.(\mathbf{go } l; x + 1) \mathbf{ in} \\ &\mathbf{let } g = \lambda x.\mathbf{print\_int } ( x * 2 ) \mathbf{ in } ( g ( f \ 3 ) ) \end{aligned}$$

If this expression is evaluated, the code of “ $(\lambda x.\text{print\_int } (x * 2)) (3 + 1)$ ” will be executed at  $l$ .

## 2.2 Contexts

A context [17] is a program expression with holes in it. The basic operation for a context is to fill its hole with an expression. For example, consider the following context.

$$\boxed{\text{let } x = 3 \text{ in } 1 + \boxed{\text{Hole}}}$$

If its hole is filled with  $(x * 2)$ , we obtain expression  $\text{let } x = 3 \text{ in } 1 + (x * 2)$ . In this expression,  $x$  in  $(x * 2)$  is bound by the **let**-construct. That is, the hole-filling operation is essentially different from *capture-avoiding* substitution in the lambda calculus [2]. Our hole-filling operation *captures* free variables. Moreover, each hole may provide a different set of bindings. The following context:

$$\boxed{\text{let } a = 1 \text{ in let } \_ = \boxed{\text{A}} \text{ in let } b = \text{true} \text{ in let } \_ = \boxed{\text{B}} \text{ in } ()}$$

will bind  $a$  and  $b$  occurring in an expression filled in “B” to 1 and true, respectively. A piece of code filled in “A” can get only the binding for  $a$ .

## 2.3 Assimilation

In our system, mobile code travels on a network, interacting with (being filled in the holes of) contexts local to nodes. A destination is designated by the name of a context and its hole. As seen in the previous subsection, contexts act as dynamic binders. Mobile code can utilize functions and data through the variables which the destination context binds. When a variable in a piece of mobile code is dynamically bound with some value by a context, we say the value is *assimilated* through the variable by the mobile code. We call such a variable an *assimilation variable* denoted by  $\backslash x$ . Assimilation is our term for dynamic binding. For example, evaluating an expression

$$\text{let } f = \lambda x.x + 1 \text{ in } (\text{go } l; (f \backslash a))$$

causes the code of “ $(f \backslash a)$ ” to migrate to the context designated by  $l$ . If the context  $l$  is

$$\boxed{\text{let } a = 3 \text{ in } \boxed{\text{Hole}}}$$

(the name of hole is omitted since the above context has a single hole) then,

$$\text{let } a = 3 \text{ in } ((\lambda x.x + 1) a)$$

will be evaluated, and the value 3 will be assimilated through  $a$  into  $(f \backslash a)$ . The variable  $f$  is bound to  $\lambda x.x + 1$  since the execution state is preserved by transparency of the migration.

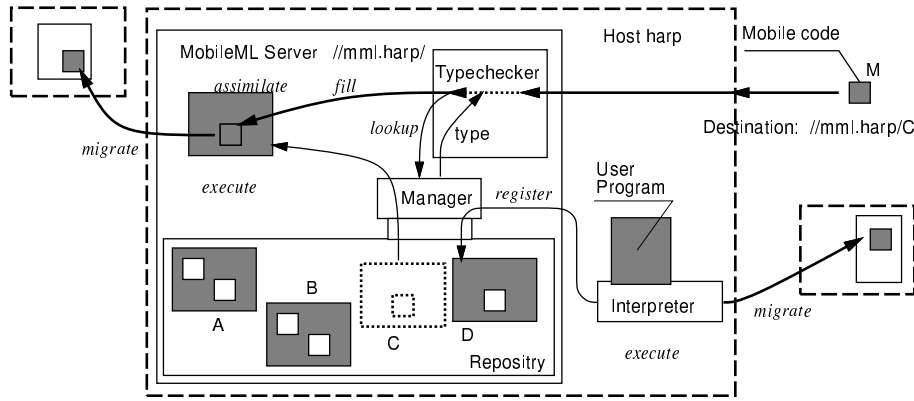


Fig. 1. MobileML system

Note that assimilation through an assimilation variable is done at the time of each migration. For example,

```
go l1; while true do go λ done
```

causes ever lasting migration as long as destination contexts exist and as long as assimilation through  $\lambda$  succeeds.

### 3 MobileML System

This section outlines the system of MobileML, and briefly introduces the syntax of MobileML.

#### 3.1 The System

Figure 1 depicts the overview of our MobileML system. The MobileML (MML for short) system consists of two parts: MML interpreter which executes user programs, and MML server which maintains contexts. The interpreter can interpret and execute programs in MML. Users can implement various mobile applications by writing programs which specify migration to contexts and registration of contexts with MML servers. Note that migration (transmission) of mobile code in MML is one-way and asynchronous like message sending in Actors [1].

The server consists of three parts: typechecker, repository, and repository manager. The typechecker checks if the type of incoming mobile code (the type information sent with the code) matches with the type of the destination context. Contexts are stored in the repository, which contains a pre-defined context named “guest” implicitly. The repository manager searches the repository for

---

$d ::= \dots$	original ML declarations
<b>context</b> $k$ <b>with</b> $X_1 \dots X_n = e$	definition of a context
$e ::= \dots$	original ML expressions
$\llbracket X \text{ hides } x_1, \dots, x_n \rrbracket$	a hole
<b>register</b> $e_1$ <b>as</b> $e_2$ <b>with</b> $e_3$	registration of a context
<b>go</b> $e$	trigger of migration
$\langle \text{assim } \{x_1, \dots, x_n\} \text{ in } e \rangle$	delimiter for migrant
$\backslash x$	assimilation variable

---

**Fig. 2.** MobileML syntax

---

the destination context and its type, or registers and stores new contexts, in response to the users' requests. In case the name of context is omitted in a search request, context "guest" is returned as the result of search. (An MML sever also includes an interpreting engine not shown in Fig. 1, in order to execute mobile code filled in a hole of a context.)

Figure 1 also shows a system snapshot where a piece of mobile code migrates to a context. In the figure,  $A$ ,  $B$ ,  $C$ , and  $D$  denote contexts named "A", "B", "C", and "D", respectively. Mobile code  $M$  is migrating to the destination indicated by the string `//mml.harp/C` where `//mml.harp` indicates an MML server running at the host machine "harp". Note that the name of hole is omitted in the destination name since  $C$  has a single hole. (In the case of multiple holes, we will write `//mml.harp/A:hole1` for instance.) Code  $M$  first arrives at the MML server, and its typechecker checks whether the type of the destination context ( $C$  in this case) and the type required by  $M$  are unifiable. For that purpose, the typechecker requests the repository manager to search the repository for the type of the context named "C". If the typechecker returns OK, the code obtained by filling  $C$ 's hole with the mobile code  $M$  is executed in the server.

### 3.2 MobileML Syntax

The syntax of MML is obtained by extending the original ML declarations and expressions with the constructs shown in Fig. 2. Contexts are defined by declaration **context**  $k$  **with**  $X_1 \dots X_n = e$ . This declaration defines a context which has body  $e$  with holes  $X_1, \dots, X_n$ , and names the context  $k$ .  $\llbracket X \text{ hides } x_1, \dots, x_n \rrbracket$  denotes a hole whose name is  $X$  and the bindings for variables  $x_1, \dots, x_n$  are *invisible* from the code to be filled in the hole  $X$ . For example, the following declaration:

```
context  $k$  with  $X =$ 
  let  $a = 1$  in let  $b = \text{true}$  in  $\llbracket X \text{ hides } a \rrbracket$ 
```

defines a context  $k$  which never binds  $a$ . When a piece of code which requires a binding for  $a$  migrates to  $X$ , a type error will occur.  $\llbracket X \text{ hides} \rrbracket$  is abbreviated

as  $\langle X \rangle$ . In order to register contexts with an MML server’s repository, we write **register**  $e_1$  **as**  $e_2$  **with**  $e_3$ . Evaluating this expression registers a context  $e_1$  as a name  $e_2$  with the repository of the server  $e_3$ .

When **go**  $e$  is evaluated, the rest of the computation runs at the destination denoted by  $e$ . Then the migrating part in the expression, which we call *migrant*, is specified by the innermost  $\langle \mathbf{assim} \{ \_ \} \mathbf{in} \_ \rangle$  which surrounds the **go** expression. We write  $\langle e \rangle$  as an abbreviation for  $\langle \mathbf{assim} \{ \_ \} \mathbf{in} e \rangle$ . For example, when we evaluate

```
 $\langle \langle \mathbf{go} \ l; \ \mathbf{print\_string} \ "Hello!" \rangle; \ \mathbf{print\_string} \ "Where \ is \ here?" \rangle,$ 
```

“Hello!” is printed at  $l$ , and “Where is here?” is printed locally. The migrant in this case is  $(\mathbf{print\_string} \ "Hello!")$ . An assimilation variable  $\_x$  appearing in a migrant gets bound at the destination. In case  $\_x$  is not bound anywhere, an exception will be raised. A delimiter  $\langle \mathbf{assim} \{x_1, \dots, x_n\} \mathbf{in} e \rangle$  determines an *assimilating migrant* for assimilation variables  $\_x_1, \dots, \_x_n$  occurring in  $e$ . That is, it specifies a migrant which is responsible for assimilation of the variables. For example, consider the following expression:

```
 $\langle \mathbf{assim} \{ \_ \} \mathbf{in} \ \mathbf{go} \ l; \ \langle \mathbf{assim} \{ a \} \mathbf{in} \ \mathbf{go} \ l'; \ \underline{\mathbf{print\_string} \ \_a} \rangle \rangle$ 
```

The assimilating migrant for  $\_a$  is the underlined part of the above expression, and assimilation through  $\_a$  occurs at the destination  $l'$  of the migrant. If we slightly modify the above expression as

```
 $\langle \mathbf{assim} \{ a \} \mathbf{in} \ \mathbf{go} \ l; \ \underline{\langle \mathbf{assim} \{ \_ \} \mathbf{in} \ \mathbf{go} \ l'; \ \mathbf{print\_string} \ \_a \rangle} \rangle,$ 
```

the assimilating migrant for  $\_a$  becomes the underlined part above. In this case, assimilation through  $\_a$  occurs at  $l$ . Delimiter  $\langle \mathbf{assim} \{x_1, \dots, x_n\} \mathbf{in} e \rangle$  is the only binder for the assimilation variables. It binds the free occurrences of  $\_x_1, \dots, \_x_n$  in  $e$ . Scoping for **assim**-binders obeys the scoping rule similar to the lambda abstraction in the lambda calculus.

By using assimilation variables, **go**, and  $\langle \mathbf{assim} \{ \_ \} \mathbf{in} \_ \rangle$ , we can describe a bit more complex migration like

```
let go_or_not =  $\lambda x$ .if test then go  $l'$  else ()
in
 $\langle \mathbf{assim} \{ a, b \} \mathbf{in}$ 
  go  $l$ ;
  print_string  $\_a$ ;
   $\langle \mathbf{assim} \{ \_ \} \mathbf{in} \ \mathbf{go\_or\_not} \ ()$ ; print_string  $\_b$ 
 $\rangle$ 
```

where “test” is a value of type **bool**. In the above code, assimilation through  $a$  and  $b$  is done at  $l$ . As the head part of the inner **assim**-construct does not declare  $b$ ,  $\_b$  in the construct is assimilated (bound) at  $l$ . So, whether “test” is “true” or “false”, assimilation through  $b$  is done at  $l$ , although “print\_string  $\_b$ ” is executed at  $l'$  when “test” is “true”.

### 3.3 Types

In addition to the types in the original ML, the set  $\tau$  of MML types contains context types specified by the following form:

$$\text{cnt}(X_1 : [F_1 \triangleright \tau_1], \dots, X_n : [F_n \triangleright \tau_n])^\tau$$

and  $F$  is written as  $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$  where each  $x_i$  denotes a variable and  $\tau_i$  is its type. A context of this type has holes  $X_1, \dots, X_n$ . We can fill  $X_i$  with an expression  $e_i$  of type  $\tau_i$ , and then the variables occurring in  $e_i$  which appear in  $F_i$  become bound. When all of the holes are filled, the context yield an expression of type  $\tau$ , which we call *result type*. The type deduction system and the type reconstruction algorithm are essentially the same as the ML type system extended with first-class contexts [16]. For example, a type of the context defined by the following declaration:

```
context  $k$  with  $X =$ 
  let  $a = 1$  in let  $b = \text{true}$  in  $\llbracket X \rrbracket$ ;
```

is automatically inferred as  $\text{cnt}(X : [\{a : \text{int}, b : \text{bool}\} \triangleright \alpha])^\alpha$ .

As mentioned in Sect. 3.1, the typechecker in an MML server dynamically checks if the type required by mobile code and the type of the destination context match. Suppose that a piece of mobile code is migrating to a context of type  $\text{cnt}(X : [F \triangleright \alpha])^\beta$  with expectation of assimilating values through variables  $x_1 : \tau_1, \dots, x_n : \tau_n$ . (The information of these variables and their types are added to the code by the system.) Then the typechecker in the MML server checks whether  $\{x_1 : \tau_1, \dots, x_n : \tau_n\} \subseteq F$  is satisfied. Since this check cannot be statically made, MML is not a strongly typed language. That is, well-typed programs in MML may cause runtime type errors. For example, the type of the following expression:

```
 $\langle \text{assim } \{a\} \text{ in go } l; \text{print\_string } \backslash a \rangle$ 
```

is inferred as  $\text{unit}$ ; however, a type error will occur at  $l$  if destination context  $l$  does not provide a value of type  $\text{string}$  for  $a$ . This is to some extent inevitable in the setting of distributed computation. However, we can cope with such kind of type error by exception-handling mechanisms.

## 4 Outline of a Semantic Model

This section outlines a semantic model for MML. As mentioned in Sect. 2, the key notions in MML are transparent migration, contexts, and assimilation. They can be described by a simple formal model, which we call  $\lambda_{mv}$ -calculus, although it is not a general model for distributed computation. Our intention here is to concentrate on the essence of MML. The issue of constructing more general models by extending  $\lambda_{mv}$ -calculus is discussed in the technical report [18].

This calculus is based on a simply typed version of Plotkin's  $\lambda_v$ -calculus [22]. The main extensions are as follows:

- representation for expressions to be evaluated concurrently by *processes* (as in CML [23]),
- the notion of first-class contexts [17], and
- representation for transparent migration.

In order to model transparent migration, we use the notion of partial continuations [9, 7]. Then, execution states of programs are abstracted as partial continuations [26, 27].

In  $\lambda_{mv}$ -calculus, a network state is represented as a set of *hosts* each of which is a pair of a host name and a set of processes. Contexts are registered with hosts (we identify MML servers with hosts in this calculus) and wait for mobile code to be filled in their holes. A piece of mobile code is transmitted from a host to a registered context by means of *tuple space* [11]. In our calculus, a tuple space acts as a virtual global buffer for mobile code transmission. That is, when a piece of mobile code is shipped off, it is first stored in the tuple space as a tuple of the destination name, the code, and the type information of what bindings the code requires. (This phase always succeeds.) Then, a context picks up the tuple whose destination and type information matches the name of the context and its type, respectively, and then fill its hole with the code in the tuple. In this way, asynchronous features of mobile code transmission in MML is modeled in our calculus.

The type system of  $\lambda_{mv}$ -calculus is based on the type system of Hashimoto and Ohoris' Context Calculus [17]. In our type system, a hole is assigned the set  $\{x_1, \dots, x_n\}$  of variables whose binders  $\lambda x_i$  surround the hole. The set is extended in the typing rule for lambda abstraction. A summary of the definition of  $\lambda_{mv}$ -calculus and its dynamic and static semantics are given in the Appendix. This calculus enjoys subject reduction lemma and type soundness theorem at the cost of runtime typecheck.

For more precise modeling of MML, we must extend the simple type system of  $\lambda_{mv}$ -calculus to a polymorphic one. We can utilize techniques elaborated for ML with first-class contexts [16] and ML with **shift/reset** [7].

## 5 Program Examples

This section gives two small examples of programming with MML. The first one is a simple example of distributed information retrieval. Applications of such a kind are often referred to as a typical example of mobile computation. The second one indicates a novel use of mobile computation: runtime update of program fragments of an application without restarting its execution.

### 5.1 News Collector

A news collector is a program which travels around a network and collects news texts. We assume that a number of contexts which provide news are already registered with MML servers running on the network. The following program

registers a news server (which is a context) named "NewsS" with an MML server running on the host "harp."

```
val top_news = ref "MobileML has been released!";

context news with READER WRITER =
  let fun get_news () = !top_news;
      val _ = [<READER>];
      fun write_news s = top_news := s;
      val _ = [<WRITTER>]
  in () end;

register news as "NewsS" with "//mml.harp/";
```

This context has two holes READER and WRITTER. It binds a variable `get_news` occurring in a piece of code to be filled in READER, and variables `get_news` and `write_news` occurring in one to be filled in WRITTER. We can write a news collector as follows:

```
fun collector () =
  let val result = ref "";
      val locs = [ "//mml.harp/NewsS:READER",
                  "//mml.lute/NewsS:READER",
                  ...
                  "//mml.banjo/NewsS:READER" ]
  in
    <|assim {get_news} in
      iter
      (fn loc =>
        go loc; result := !result ^ (get_news()) ^ "\n"
      ) locs;
      go "//mml.halle/MyHome";
      print_string !result
    |>
  end;
```

where `iter` is the usual iteration function of type  $(\alpha \rightarrow \text{unit}) \rightarrow \alpha \text{ list} \rightarrow \text{unit}$ . A migrant is created by evaluating "collector ()", and travels around the destinations given in the list `locs` in order to collect the news. A reference cell `result` is used to accumulate the items. The result will be displayed at "`//mml.halle/MyHome`". Note that reference cells are always copied and do not create a remote reference in MML.

## 5.2 Runtime Library Update

Usually, an application software is constructed so that necessary libraries are linked only at starting time. If we want to try a new version of a library, we must reboot the application. Rebooting an application would be very irritating if it

needs much time. In our scheme of “runtime library update”, an application (or its main program) is represented by a piece of mobile code which uses a library and the library is considered as a context. The following programs illustrate our scheme.

```

val lib_name = ref "";
context lib1 with APPL =
  let val new_lib = lib_name;
      val f1 = ...;
      val f2 = ...;
      ...
      val fn = ...
  in [<APPL>] end;

```

```

register lib1 as "lib-v1" with "//mml.harp/";

```

The above code defines and registers a context representing a library. A piece of code migrating to hole APPL can use functions  $f_1, \dots, f_n$  in the library. In this setting, an application starts its execution by migrating to such a context. An application is defined as follows:

```

fun run () =
  <|assim {new_lib} in
    go "//mml.harp/lib-v1";
    while true do
      if !'new_lib <> "" then go !'new_lib else do_jobs();
    done
  |>;

```

By evaluating “run ()”, a migrant migrates to the library lib1, and repeats the execution of “do\_jobs ()” while the content of new\_lib is “”.

Meanwhile, a developer of the library completes a new version lib2 of the library. He distributes the new version using a function named updator given below.

```

val lib_name = ref "";
context lib2 with APPL =
  let val new_lib = lib_name;
      val f1 = ...;
      val f2 = ...;
      ...
      val fn = ...
  in [<APPL>] end;

fun updator () =
  <|go "//mml.harp/";
    register lib2 as "lib-v2" with "//mml.harp/";
    <|assim {new_lib} in
      go "//mml.harp/lib-v1"; 'new_lib:= "//mml.harp/lib-v2"
    |>
  |>;

```

By applying () to `updater`, a migrant (1) migrates to the server at “harp”, (2) registers the new version with the server, (3) migrates to the old library, (4) lets the application know the name of the new version, by updating the content of shared variable `new_lib` to the (full) name of the new version. Finally, the application becomes aware of the name of the new version in the `while`-loop, and migrates to the new version of the library. Due to transparency of migration, the application can continue to run with the new library without being rebooted.

## 6 Related Work

Facile [20] supports a mechanism similar to that of our “assimilation”. It is a concurrent distributed language based on a higher-order strongly typed language. This language supports closure transmission similar to that of Obliq [4]. The feature of this language that we should pay attention to is a mechanism for dynamic binding with remote resources. However, that is slightly different from our mechanism. Facile uses *proxy structures* in order to specify the variables which become bound at a remote site. A proxy structure is generated from a *remote signature* which specifies names and types of the values provided at the remote site. We can treat proxy structures as if they were local structures. That is, by means of proxy structures, a closure can refer the remote structures which match the remote signature at the compilation time. When a closure arrives at the destination, the closure is linked with the structures specified by remote signatures, assuming that such structures always exist. Therefore, in case specifications of remote structures are modified, it is necessary to get the new remote signatures and to recompile programs which refer the structures. Our MML does not need such recompilation since such consistency is dynamically checked.

## 7 Concluding Remarks

Mobile computation has the advantages of high efficiency and productivity in developing novel distributed applications such as workflow systems, flexible software distribution, and intelligent search systems. In our current work, we have designed a language which facilitates programming for mobile computation. The innovative features of our language include the notion of contexts which allows us to succinctly describe the interaction between mobile code and environments at destination nodes. Since our language is based on ML, we can expect that theoretical developments and implementation techniques elaborated for ML are readily applicable to our system. We have also developed a semantic model for our language based on CML-style semantics and tuple spaces. In our model, tuple spaces act as virtual global buffers for mobile code transmission. This model enjoys type soundness theorem assuming that we can always determine at runtime if the type of a migrant matches the type of the destination context.

There are several interesting issues that merit further investigation. We briefly mention some of them.

*Communication between migrants.* In the current MML system, migrants running at the same context can communicate by using reference cells visible in the context. However, special mechanisms for communication between migrants are not provided. If we extend MML with a communication mechanism using channels as found in CML [23] or *transiently shared tuple spaces* of LIME [21], the convenience will increase especially in case of inter-migrant communication in the same context or host machine.

*Security.* In the present system, visibility of resources from mobile code can naturally be controlled by means of **hide** and multiple different holes. However, security issues such as authentication and verification of mobile code are not addressed yet. For example, we would like to incorporate into our system theoretical results such as prevention of security violation by SLam Calculus [19], and type-based specification mechanisms of access control policies of KLAIM [8] or  $D\pi$  [24].

*Efficient implementation.* We have implemented an experimental interpreter system on OCaml. However, there is plenty of room for improvements to make our system be suitable for the practical use. We are planning to construct a bytecode based compiler system. Techniques for closure transmission developed in Facile [20] will be applicable to our code transmission.

## Acknowledgments

The authors would like to acknowledge fruitful discussions with the members of the AMO Project.

## References

1. Gul Agha. *ACTORS : A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. revised edition.
3. Krishna A. Bharat and Luca Cardelli. Migratory Applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Pittsburgh, Pa., November 1995. Also available as Digital Systems Research Center Research Report 138.
4. Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Also available as Digital Systems Research Center Research Report 122.
5. Luca Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science, pages 3–6. Springer-Verlag, April 1997.
6. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
7. Olivier Danvy and Andrzej Filinski. A Functional Abstraction of Typed Contexts. Technical report, Institute of Datalogy, University of Copenhagen, 1989. DIKU 89/12.

8. R. De Nicola, G. Ferrari, and R. Pugliese. Types as Specifications of Access Policies. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 117–146. Springer-Verlag, 1999.
9. M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps. In *ACM Conference on Lisp and Functional Programming*, pages 52–62, 1988.
10. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996, Proceedings)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
11. David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
12. J. Gosling and H. McGilton. The Java Language Environment. White paper, Sun Microsystems, 1995.
13. Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., December 1995.
14. Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agent Security*, Lecture Notes in Computer Science, pages 154–187. Springer-Verlag: Heidelberg, Germany, 1998.
15. Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Research report, IBM T. J. Watson Research Center, March 1995.
16. Masatomo Hashimoto. First-Class Contexts in ML. In *Proc. 4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 206–223. Springer, 1998.
17. Masatomo Hashimoto and Atsushi Ohori. A Typed Context Calculus. Preprint 1098, Research Institute for Mathematical Sciences, Kyoto, Japan, 1996. Revised version to appear in TCS.
18. Masatomo Hashimoto and Akinori Yonezawa. A Typed Language for Mobile Computation. Technical report, Univ. Tokyo, 2000. In preparation.
19. Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 365–377, New York, NY, January 1998. ACM.
20. Frederick C. Knabe. An overview of mobile agent programming. In *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Stockholm, Sweden, June 1996.
21. Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377. ACM Press, May 1999.
22. G. Plotkin. Call-by-name, call-by-value, and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
23. J. H. Reppy. CML: A higher-order Concurrent Language. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

24. James Riely and Matthew Hennessy. A Typed Language for Distributed Mobile Processes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 378–390. ACM Press, January 1998.
25. Tatsuro Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Proceedings of the Third International Conference on Coordination Models and Languages*, 1999.
26. Tatsuro Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In H. Bowman and J. Derrick, editors, *Proceedings of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 21–36. Chapman&Hall, 1997.
27. Takuo Watanabe. Mobile Code Description using Partial Continuations: Definition and Operational Semantics. In *SWoPP '97*, pages 61–66. IPSJ, August 1997.
28. James E. White. Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. The MIT Press, 1996.

## A Summary of $\lambda_{mv}$ -Calculus

In this appendix, we briefly introduce  $\lambda_{mv}$ -calculus – a simply typed lambda calculus with the features of mobile computation in MML. This calculus is based on a simply typed version of Plotkin’s  $\lambda_v$ -calculus [22]. A dynamic semantics of this calculus is obtained by defining a transition system between *configurations* which represent network states. A configuration is a pair of a *tuple space* [11] and a set of *hosts*. The tuple space acts as a virtual global buffer for code transmission. A host is a pair of host name and a set of processes (i.e., named programs which are evaluated locally and concurrently).

We use the following notation for sets. Let  $A$  and  $B$  be sets. Then  $A \cup B$  is their union,  $A \uplus B$  is their disjoint union, and  $A \setminus B$  is their difference. We sometimes omit braces for singletons, for instance, we use  $\{a, b\} \uplus c$  for  $\{a, b\} \uplus \{c\}$ . Let  $M$  and  $N$  be multisets. Then  $M \oplus N$  denotes their union.

### A.1 Types and Expressions

The set of types (ranged over by  $\tau$ ) of  $\lambda_{mv}$  is given by the following syntax:

$$\tau ::= b \mid \tau \rightarrow \tau \mid \text{cnt}_\rho$$

where  $b$  stands for a given set of base types,  $\text{cnt}_\rho$  for *context types*, and  $\rho$  for *assimilation variable sets*  $\{a_1^{\tau_1}, \dots, a_n^{\tau_n}\}$  of *assimilation variables*  $a_i$  annotated with their types  $\tau_i$ . We assume that each context has a single hole of type unit, and its result type is unit. Thus,  $\text{cnt}_\rho$  suffices for context type.

Let  $x^{a:\tau}$  range over a countably infinite collection of variables labeled with their types and assimilation variables where  $a$  on the shoulder of  $x$  denotes the external name of  $x$ . And let  $X$  range over a countably infinite collection of holes. The sets of values ranged over by  $v$  and expressions ranged over by  $e$  are given

by the following syntax:

$$v ::= c^\tau \mid a^\tau \mid x^{a:\tau} \mid \lambda x^{a:\tau}.e \mid X \mid \delta X : \rho.e$$

$$e ::= v \mid e e \mid \langle e \rangle_\rho \mid \mathbf{go}(e, e) \mid \mathbf{reg}(e, e)$$

where  $\delta X : \rho.e$  stands for contexts, and  $c^\tau$  for a given set of constants labeled with their types. The set of constants contains host names  $h^{\text{hstn}}$  and process names  $p^{\text{procn}}$ . We write  $\langle e \rangle$  as an abbreviation for  $\langle e \rangle_{\{\}} \mid \{\}$ . The set  $\text{FV}(e)$  of free variables in  $e$  is defined in the usual way, according to the structure of  $e$ . The following clauses are examples.

$$\begin{aligned} \text{FV}(x^{a:\tau}) &= \{x^{a:\tau}\} & \text{FV}(\lambda x^{a:\tau}.e) &= \text{FV}(e) \setminus \{x^{a:\tau}\} \\ \text{FV}(\delta X : \rho.e) &= \text{FV}(e) & \text{FV}(\langle e \rangle_{\{a_1^{\tau_1}, \dots, a_n^{\tau_n}\}}) &= \text{FV}(e) \end{aligned}$$

The set  $\text{FAV}(e)$  of free assimilation variables in  $e$  is defined similarly. The following clauses are also examples.

$$\begin{aligned} \text{FAV}(a^\tau) &= \{a^\tau\} & \text{FAV}(\lambda x^{a:\tau}.e) &= \text{FAV}(e) \\ \text{FAV}(\delta X : \rho.e) &= \text{FAV}(e) & \text{FAV}(\langle e \rangle_\rho) &= \text{FAV}(e) \setminus \rho \end{aligned}$$

We denote the set of free holes in  $e$  by  $\text{FH}(e)$ . We can safely assume  $\alpha$ -renaming of bound holes just as bound variables in the lambda calculus. By  $e\{x^{a:\tau} \mapsto e'\}$  and  $e\{a^\tau \mapsto e'\}$ , we denote the expressions obtained by substituting  $e'$  for any free occurrence of  $x^{a:\tau}$  in  $e$  and  $a^\tau$  in  $e$ , respectively. The notion of  $\alpha$ -congruence in  $\lambda_{mv}$  is defined as:

$$\lambda x^{a:\tau}.e \equiv_\alpha \lambda y^{a:\tau}.e\{x^{a:\tau} \mapsto y^{a:\tau}\} \quad (\text{if } y \notin \text{FV}(e))$$

## A.2 Type System

A *hole type assignment*, ranged over by  $\Delta$ , is  $\emptyset$  or a singleton of a pair of a hole and its assimilation variable set. The type system of  $\lambda_{mv}$  is defined as a proof system to derive a *typing judgment* of the form:

$$\Delta \vdash e : \tau$$

where expression  $e$  has type  $\tau$  under hole type assignment  $\Delta$ . The set of typing rules is given in Fig. 3. In type inference, we do not apply the inference rules which add redundant elements to the hole type assignments.

## A.3 Local Evaluation

We first define an evaluation relation “ $\longrightarrow$ ” local to processes, following the line of  $\lambda_v$ . An evaluation context is an expression containing a single “[ ]” which marks the next redex. Evaluation contexts are standard evaluation contexts for call-by-value evaluation, and are defined by the following syntax:

$$E ::= [ ] \mid E e \mid v E \mid \langle E \rangle_\rho$$

---

$\text{Cst } \emptyset \vdash c^\tau : \tau$	$\text{Asv } \emptyset \vdash a^\tau : \tau$	$\text{Var } \emptyset \vdash x^{a:\tau} : \tau$
$\text{Fun } \frac{\{X : \rho\} \vdash e : \tau'}{\{X : \rho \uplus \{a : \tau\}\} \vdash \lambda x^{a:\tau}.e : \tau \rightarrow \tau'}$	$\text{App } \frac{\Delta_1 \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta_2 \vdash e_2 : \tau_1}{\Delta_1 \uplus \Delta_2 \vdash e_1 e_2 : \tau_2}$	
$\text{Hol } \{X : \{\}\} \vdash X : \text{unit}$	$\text{Cnt } \frac{\{X : \rho\} \vdash e : \text{unit}}{\emptyset \vdash \delta X : \rho.e : \text{cnt}_\rho}$	
$\text{Go } \frac{\Delta_1 \vdash e_1 : \text{hstn} \quad \Delta_2 \vdash e_2 : \text{procn}}{\Delta_1 \uplus \Delta_2 \vdash \text{go}(e_1, e_2) : \text{unit}}$	$\text{Mig } \frac{\Delta \vdash e : \text{unit}}{\Delta \vdash \langle e \rangle_{\{a_1^{\tau_1}, \dots, a_n^{\tau_n}\}} : \text{unit}}$	
$\text{Reg } \frac{\Delta_1 \vdash e_1 : \text{cnt}_\rho \quad \Delta_2 \vdash e_2 : \text{procn}}{\Delta_1 \uplus \Delta_2 \vdash \text{reg}(e_1, e_2) : \text{unit}}$		

---

**Fig. 3.** Typing rules

---

The local evaluation relation “ $\longrightarrow$ ” is the smallest relation satisfying the following rule:

$$\begin{array}{ll}
(\text{prim}) & E[c^\tau v] \longrightarrow E[\text{Prim}(c^\tau, v)] \\
(\beta) & E[(\lambda x^{a:\tau}.e) v] \longrightarrow E[e\{x \mapsto v\}] \quad (\text{FH}(e) = \text{FH}(v) = \emptyset) \\
(\text{migr}) & E[\langle v \rangle_\rho] \longrightarrow v
\end{array}$$

where “Prim” denotes a partial function which gives meanings to the function constants. We assume that if  $\emptyset \vdash v : \tau$ , then  $\text{Prim}(c^{\tau \rightarrow \tau'}, v)$  is defined and  $\emptyset \vdash \text{Prim}(c^{\tau \rightarrow \tau'}, v) : \tau'$ .

#### A.4 Global Evaluation

Global evaluation is defined by a transition system of configurations. This is similar to the style of CML [23]. The global evaluation relation “ $\Longrightarrow$ ” extends “ $\longrightarrow$ ” to configurations, and adds additional rules for context registration and migration. We denote by  $[e]^p$  a process which is named  $p$  and executes  $e$ . A host containing its name  $h$  and a set  $\mathcal{P}$  of processes is denoted by  $\langle h, \mathcal{P} \rangle$ . We denote a set of hosts by  $\mathcal{H}$ . A migrant is denoted by  $[h, p, e, \rho]$ , and a migrant set is a multiset (tuple space) of migrants ranged over by  $\mathcal{M}$ . We denote a configuration by  $\mathcal{M}, \mathcal{H}$ .

A process set  $\mathcal{P}$  is well-formed if for all  $[e]^p \in \mathcal{P}$ , the following hold:  $\text{FV}(e) = \text{FAV}(e) = \emptyset$  and there is no  $e' \neq e$  such that  $[e']^p \in \mathcal{P}$ . A host set  $\mathcal{H}$  is well-formed if for all  $\langle h, \mathcal{P} \rangle \in \mathcal{H}$ , the following hold:  $\mathcal{P}$  is well-formed, and there is no  $\mathcal{P}' \neq \mathcal{P}$  such that  $\langle h, \mathcal{P}' \rangle \in \mathcal{H}$ . A migrant set  $\mathcal{M}$  is well-formed if for all  $[h, p, e, \rho] \in \mathcal{M}$ ,  $\text{FV}(e) = \emptyset$ . A configuration  $\mathcal{M}, \mathcal{H}$  is well-formed if both  $\mathcal{M}$  and  $\mathcal{H}$  are well-formed.

A *process typing*  $\text{PT}$  is a finite map from long process names (each of which is a pair of a host name and a process name) to types. A well-formed configuration

$\mathcal{M}, \mathcal{H}$  has type  $\text{PT}$ , denoted by  $\vdash \mathcal{M}, \mathcal{H} : \text{PT}$ , if the following hold:  $\{(h, p) \mid h \in \text{dom}(\mathcal{H}) \text{ and } p \in \text{dom}(\mathcal{H}(h))\} \subseteq \text{dom}(\text{PT})$ , for all  $e$  such that  $[e]^p \in \mathcal{P}$  and  $\langle h, \mathcal{P} \rangle \in \mathcal{H}$ ,  $\emptyset \vdash e : \text{PT}(h, p)$ , and for all  $[h, p, e, \rho] \in \mathcal{M}$ ,  $\emptyset \vdash e : \text{unit}$  and  $\text{FAV}(e) \subseteq \rho$ .

The global evaluation relation is defined by four inference rules each of which defines a single step evaluation. The first rule simply extends the local evaluation relation to configurations:

$$\text{Exp} \frac{e \longrightarrow e'}{\mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [e]^p \rangle \Longrightarrow \mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [e']^p \rangle}$$

In the following rule, context  $\delta X : \rho.e'_1$  is registered as  $p$  with  $h$ .

$$\text{Reg} \frac{e_1 \longrightarrow \delta X : \rho.e'_1 \quad e_2 \longrightarrow p}{\mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [E[\mathbf{reg}(e_1, e_2)]] \rangle \Longrightarrow \mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [E[()]] \uplus [\delta X : \rho.e'_1]^p \rangle}$$

In the following,  $R[-]$  denotes an evaluation context which does not contain any  $\langle \_ \rangle$ . The expression  $R[()]$  is packed in a tuple, which is added (**out**-ed) to  $\mathcal{M}$ .

$$\text{Go} \frac{e_1 \longrightarrow h \quad e_2 \longrightarrow p}{\mathcal{M}, \mathcal{H} \uplus \langle h', \mathcal{P} \uplus [E[\langle R[\mathbf{go}(e_1, e_2)] \rangle_\rho]] \rangle \Longrightarrow \mathcal{M} \oplus [h, p, R[()], \rho], \mathcal{H} \uplus \langle h', \mathcal{P} \uplus [E[()]] \rangle}$$

We denote by  $e\{e'/X\}$  an expression obtained from  $e$  by filling its hole  $X$  with  $e'$ . Let  $X$  occur in the scopes of  $\lambda x_1^{a_1:\tau_1}, \dots, \lambda x_n^{a_n:\tau_n}$  in  $e$ . Then we define *renamer*  $\theta_e^{X:\{a_1^{\tau_1}, \dots, a_n^{\tau_n}\}}$  as substitution  $\{a_1^{\tau_1} \mapsto x_1^{a_1:\tau_1}, \dots, a_n^{\tau_n} \mapsto x_n^{a_n:\tau_n}\}$ . A migrant can access local resources provided by a context through assimilation variables, by applying the migrant to the renamer before hole-filling. The rule below shows that a migrant matching with a context can arrive at the context.

$$\text{Fill} \frac{\rho' \subseteq \rho \quad p' \text{ is fresh}}{\mathcal{M} \oplus [h, p, e', \rho'], \mathcal{H} \uplus \langle h, [\delta X : \rho.e]^p \uplus \mathcal{P} \rangle \Longrightarrow \mathcal{M}, \mathcal{H} \uplus \langle h, [e\{\theta_e^{X:\rho}(e')\}/X]^{p'} \uplus [\delta X : \rho.e]^p \uplus \mathcal{P} \rangle}$$

A process  $[e]^p$  is *stuck* if  $e$  is not a value and there do not exist well-formed configurations  $\mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [e]^p \rangle$  and  $\mathcal{M}', \mathcal{H}'$  such that  $\mathcal{M}, \mathcal{H} \uplus \langle h, \mathcal{P} \uplus [e]^p \rangle \Longrightarrow \mathcal{M}', \mathcal{H}'$ , with  $p$  a selected process of this transition. A well-formed configuration is *stuck* if one or more of its processes are stuck.

We have the following lemma and theorem. The proof is given in the technical report [18].

**Lemma 1 (Subject Reduction).** *If a configuration  $\mathcal{M}, \mathcal{H}$  is well-formed,  $\mathcal{M}, \mathcal{H} \Longrightarrow \mathcal{M}', \mathcal{H}'$ , and  $\vdash \mathcal{M}, \mathcal{H} : \text{PT}$ , then there exists a process typing  $\text{PT}'$  such that  $\text{PT} \subseteq \text{PT}'$ ,  $\vdash \mathcal{M}', \mathcal{H}' : \text{PT}'$ , and  $\vdash \mathcal{M}, \mathcal{H} : \text{PT}$ .*

**Theorem 1 (Type Soundness).** *Well-formed configurations do not get stuck.*