Mobile Reactive Programming in ULM

Stéphane Epardaud
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Stephane.Epardaud@sophia.inria.fr

Abstract

We present the embedding of ULM [7] in Scheme and an implementation of a compiler and virtual machine for it. ULM is a core programming model that allows multi-threaded and distributed programming via strong mobility with a deterministic semantics. We present the multi-threading and distributed primitives of ULM step by step using examples. The introduction of mobility in a Scheme language raises questions about the semantics of variables with respect to migration. We expose the problems and offer two solutions alongside ULM's network references. We also present our implementation of the compiler, virtual machine and the concurrent threading library written in Scheme.

1 Introduction

Today's networks of computers have nothing to do with what we had twenty years ago. While there were very few of them back then, it is now very hard not to be surrounded by more than one computer, practically always connected to some sort of network. And if networks and computers have drastically evolved and multiplied, it is natural that programming languages evolve to exploit their number and interconnections.

The widespread clustering of processors have marked the appearance of parallel multi-threading, while the connectivity phenomenon has brought along distributed programming. Some programming languages nowadays include these features right alongside the `+` and `set!` operations.

However, there are many ways to do multi-threading, and parallel execution is but one of them. Many people accept the common idea of preemptive non-deterministic scheduling and the variety of problems that are bundled along. Deadlocks, race conditions and synchronisation problems are but a few problems that one experiences while taking the perilous learning experience of what we often call *native* threads. Debugging a non-deterministic program is a challenging feat, especially since running it on a single processor does not help.

Reactive programming offers the ability to execute multi-threading programs in a concurrent and deterministic way. No more fancy scheduling, surprising race conditions and synchronisation mysteries. Every execution of an otherwise multi-threading program runs according to a precise semantics: the same level of predictability expected from `+` or `set!`.

Distributed programming, much like multi-threading programming has many variants, all of them bearing the unreliability of networks. From communication breakdowns to computer unavailability, the networking part of distributed computing faces non-determinism much in the same way parallel threads do. If distributed programming techniques are to be incorporated into a deterministic language, there has to be a way to isolate any non-determinism in safe and well-known places.

ULM (*Un Langage pour la Mobilité*) is a set of reactive and distributed primitives written by G. Boudol that offers a deterministic semantics for local execution. Following the GALS (Globally Asynchronous Locally Synchronous) model, ULM offers concurrent deterministic multi-threading on each site, together with strong thread mobility while isolating non-determinism.

This paper presents our implementation of a prototype interpreter for a Scheme language augmented with ULM primitives. We introduce the syntax of this new language along with illustrations of how to use the ULM constructs in Scheme. We present the implications of introducing mobility in a Scheme language and how we chose to address them in a way that fits with ULM's objectives. Rather than detail the implementation of our working prototype, we explain the global ideas behind the cooperative scheduling and the mobility of threads.

In Section 2 we present the reactive aspect of ULM. In Section 3 we introduce the mobility primitives, the problems arising from free variables during migration, and the different types of variables we offer as solution. In Section 4 we present an extended example of mobility and multi-threading through agent interactions. The implementation of our prototype compiler and virtual machine is outlined in Section 5. We muse on future directions in Section 6, compare our implementation with related work in Section 7 and finally conclude in Section 8.

2 ULM Reactive Primitives

ULM is inspired by FairThreads [6], which is a reactive variant of Esterel [8], an imperative synchronous language, but you need not

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA. Copyright 2004 INRIA.

know these languages in order to understand ULM and no prior knowledge will be assumed in this paper.

In reactive programs, execution time is divided in units called *instants*. An instant is a discrete time interval during which threads are allowed to react. The basic idea is that during an instant, all threads that want to run are allowed to run, until they all decide to wait for the next instant (a form of cooperation), or are blocked while waiting for something that is not going to happen during the instant. When all threads are waiting or blocked, we go to the next instant.

2.1 Threads

In order to present the ULM language, we will show some examples of what you can do with it. First of all, we will present the basics of multi-threading in ULM. Its threads are lightweight, cooperative, not parallel but concurrent, do not have their own memory space, and are scheduled in a deterministic fashion. They resemble *event loop* programs, that do not need locks for synchronisation, but need to cooperate at some points to let the other threads run.

Here are two threads that run concurrently (the example is explained below):

```
1: (define (make-printer-thread name)
2:   (ulm:thread
3:     (lambda ()
4:       (let loop ()
5:         (print name)
6:         (ulm:pause)
7:         (loop)))))
8: (make-printer-thread "A")
9: (make-printer-thread "B")
```

In this example we define a procedure (`make-printer-thread`) that creates a new thread that will print its name, cooperate with other threads, and keep doing that forever. We then (lines *8-9*) use that procedure to create two threads: one that will print "A" and the other "B". The result is that "A" and "B" will be printed repeatedly in that order forever.

The procedure *ulm:thread* takes a thunk as parameter, creates a thread that will execute this thunk, and schedules the thread to be run later at the current instant. This program also illustrates that the toplevel execution is in an implicit thread: when the toplevel has finished execution, it implicitly terminates and lets other threads run (in this case, the threads that print "A" and "B"). The concept of cooperation is illustrated in this example with the call to *ulm:pause*, whose effect is to wait for the next instant, explicitly allowing other threads to execute. Calling *ulm:pause* after each thread prints allows the other threads to execute, and makes sure the calling thread won't be awakened until all the other threads are done for this instant, which is when the next instant starts.

2.2 Signals

This example and its following explanation introduce the concept of inter-thread communication:

```
1: (let ((relay (ulm:signal)))
2:   (ulm:thread
3:     (lambda ()
4:       (print "Thread A starts and waits for B")
5:       (ulm:await relay)
6:       (print "Thread A resumes execution")))
7:   (ulm:thread
8:     (lambda ()
9:       (print "Thread B starts and wakes up A")
10:      (ulm:emit relay)
11:      (print "Thread B now terminates"))))
```

In this example, we create a *signal* (*ulm:signal* returns a new signal at line *1*) called `relay` which will serve as a synchronisation and communication means between threads. A signal is a sort of flag that is set to "not there" at the beginning of each instant, and then is set to "present" (or any value, as we will see later) as soon as someone emits it, until the next instant. This allows threads to wait for a signal to be emitted (if it is not present already), and to emit signals to wake up other threads. Here thread `A` starts executing[1], then waits for the `relay` signal (line *5*) to be emitted, which implicitly allows thread `B` to run (since thread `A` is waiting). Thread `B` then runs, and emits `relay` (line *10*), which implicitly allows thread `A` to be awakened and rescheduled later in the instant. Then thread `B` terminates and execution passes to thread `A` (line *6*), which then terminates.

Communication of values between threads is done in a classical manner, through shared variables and synchronisation is ensured via signals. The next version of our ULM interpreter will have an object system in the form of mixins [9], and provide an *Event* mixin that will serve as a signal with an associated value. The next section will discuss the differences in communicating values between threads when migration is involved.

2.3 Suspension

The first of the two main reactive primitives in ULM is *ulm:when*, which introduces suspension. Suspension causes a program to be suspended at each instant when a signal has not been emitted (to put it differently: it allows a program to run only during instants in which a signal is emitted).

We will now illustrate and explain suspension:

```
1: (let ((odd-signal (ulm:signal))
2:       (even-signal (ulm:signal)))
3:   (ulm:thread
4:     (lambda ()
5:       (let loop ((even #t))
6:         (if even
7:           (ulm:emit even-signal)
8:           (ulm:emit odd-signal))
9:         (ulm:pause)
10:        (loop (not even)))))
11:  (ulm:thread
12:    (lambda ()
13:      (ulm:when odd-signal
14:        (lambda ()
15:          (let loop ()
16:            (print "Odd instant")
17:            (ulm:pause)
18:            (loop)))))))
```

This program creates two signals: `odd-signal` and `even-signal`

[1] Because of the deterministic semantics of scheduling, threads are run in their creation order.

that will serve respectively for marking odd and even instants. The first thread loops forever and emits alternatively the `even-signal` or the `odd-signal` at each instant. The second thread enters suspension (line *13*): that is, it is allowed to run only when `odd-signal` is present. At the beginning of each instant it will be blocked until `odd-signal` has been emitted, and only then will it be allowed to run until the next instant. As soon as `odd-signal` has been emitted, the second thread prints something and waits for the next instant (lines *16*-*17*), which means it will be blocked again until `odd-signal` is emitted again.

2.4 Preemption

The second main reactive primitive is weak preemption, which allows a program to be given up at the end of an instant. Here we introduce preemption and explain it below:

```
 1: (define (run-n-instants n thunk)
 2:   (let ((kill-signal (ulm:signal)))
 3:     (ulm:thread
 4:       (lambda ()
 5:         (let loop ((i n))
 6:           (if (> i 0)
 7:             (begin
 8:               (ulm:pause)
 9:               (loop (- i 1)))
10:             (ulm:emit kill-signal)))))
11:     (ulm:watch kill-signal thunk)))
12:
13: (run-n-instants 3
14:   (lambda ()
15:     (let loop ()
16:       (print "New instant")
17:       (ulm:pause)
18:       (loop))))
```

This example defines a procedure (`run-n-instants`) which takes a `thunk` and a number of instants `n` as arguments. This procedure creates a thread that will wait for `n` instants (lines *6*-*9*) and then emit a `kill-signal` (line *10*). Before that thread even starts, the procedure will enter a preemption block on `kill-signal` in which it will execute `thunk` (line *11*). The effect of this is that `thunk` will be allowed to run during at most `n` instants because then the killer thread will emit the `kill-signal` which will cause the execution of `thunk` to be aborted at the end of the n^{th} instant.

We then call (line *13*) `run-n-instants` in the implicit thread, with a procedure that prints each new instant. The result is that the implicit thread will print three instants and then return from `run-n-instants`.

There is a minor problem with this code though, because the *ulm:when* and *ulm:watch* blocks terminate immediately upon termination of their body, whether or not the preemption or suspension signal has been emitted. This means that once `thunk` returns (say, a normal return, not interrupted by the `kill-signal`), the killer thread is still running for a number of instants, albeit harmlessly, since emitting the `kill-signal` after `thunk` has returned means that there is nothing to preempt anymore. This is still a waste of execution and a more proper way to implement `run-n-instants` would be such:

```
 1: (define (run-n-instants n thunk)
 2:   (let ((kill-signal (ulm:signal))
 3:         (done (ulm:signal)))
 4:     (ulm:thread
 5:       (lambda ()
 6:         (ulm:watch done
 7:           (lambda ()
 8:             (let loop ((i n))
 9:               (if (> i 0)
10:                 (begin
11:                   (ulm:pause)
12:                   (loop (- i 1)))
13:                 (ulm:emit kill-signal)))))))
14:     (ulm:watch kill-signal thunk)
15:     (ulm:emit done)))
```

In ULM, preemption is said to be *weak* because it does not happen at the moment the preemption signal is emitted during the instant, but only at the end of the instant. This is different from the common notion of preemption using exceptions where no code is executed between the throwing and the raising (or even from *strong* preemption in ESTEREL [8] where the preempted code is not even executed in the first place). In ULM a preempted thread can continue executing at most until the end of the instant. This delaying of preemption (and migration, as we will see later) to the end of instant, as opposed to during the instant, happens because control flow within a given instant should be independent of the scheduling order.

With only the primitives *ulm:thread, ulm:signal, ulm:emit, ulm:when* and *ulm:watch*[2] we can create threads, allow them to cooperate, let them enter critical sections, make them communicate and decide how they should be sequenced.

3 ULM Mobility

ULM defines *strong* mobility primitives that allow *agents* to migrate between *sites*. Strong mobility allows threads to be migrated with their state, without special treatment from the programmer. This type of migration is transparent and the migrated thread doesn't need to notice it migrated. Note the strong and weak variants of preemption are different from the strong or weak notions in mobility. In our implementation, a site is a Virtual Machine (VM), be it on the same computer or on separate networks.

3.1 Mobility

In ULM, an agent is a kind of thread that can move across sites and has an *agent heap*. There are two kinds of heaps in ULM: the classical heap (called *site heap*) is local to a site and does not move. Agent heaps on the other hand are attached to agents, and migrate with them. A variable allocated in either heap is accessed by means of a *reference*, which can be *local* if the reference is on the same site as the heap it points to, or *remote* if they are on different sites. A local reference access is non-blocking while a remote reference access is always blocking. Whenever a thread attempts to access a reference whose heap is remote, it will be blocked until that heap comes in (with its agent) or until the blocked thread is migrated to the heap's site, unless the access is preempted, of course.

Just like a thread is created with (*ulm:thread* `thunk`), we create an agent with (*ulm:agent* `proc`). Migration of an agent is *strong*, and so an agent migrates with its code, registers, stack and heap. Migration is done by calling either (*ulm:migrate-to*

[2]Implementation of *ulm:await* and *ulm:pause* using these primitives is left as an exercise for the reader.

host) or (*ulm:migrate-to* host agent). The first form migrates the current agent to the given host, while the second form migrates the given agent (*subjective* and *objective* migration resp.). While *ulm:thread* takes a thunk for the thread body, *ulm:agent* takes a procedure of one argument for the agent's body. This procedure will be called with the agent's name, which is also returned by the *ulm:agent* call to the creating thread. This name is used for objective migration.

As we mentioned earlier, migration only happens between instants (like preemption), and (*ulm:migrate-to* host) does not block until the end of instant. This means that anything executed between the call to *ulm:migrate-to* and the end of instant will be executed *prior* to moving. This is why we often use *ulm:pause* after *ulm:migrate-to*. The reason why subjective migration is non-blocking is to keep it symmetrical with objective migration, which has no reason to be blocking.

Here is a first example of migration:

```
1: (ulm:agent
2:   (lambda (name)
3:     (print "here")
4:     (ulm:migrate-to "other-host")
5:     (ulm:pause)
6:     ;; we are now on 'other-host'
7:     (print "there")))
```

This creates an agent that will print something on its creation site, then migrate (line *4*) and wait for arrival (line *5*). Once it arrives on the new site, it prints something there and terminates.

3.2 Mobility Groups

There is a certain hierarchy between agents and threads: threads have a parent, which can be either the local site, or an agent. Agents on the other hand do not have a parent. Any thread created directly by the implicit thread has the local site as parent, while any thread created directly or indirectly by an agent has that agent as parent. This allows us to form groups of threads that will function and migrate together. Let us illustrate migration grouping:

```
 1: (ulm:thread
 2:   (lambda ()
 3:     (print "Our parent is the local site")
 4:     ; create a new agent and store its name
 5:     (let ((name
 6:            (ulm:agent
 7:              (lambda (name)
 8:                ; create a thread that
 9:                ; stays with us
10:                (ulm:thread
11:                  (lambda ()
12:                    (let loop ()
13:                      (print "Second thread")
14:                      (ulm:pause)
15:                      (loop))))
16:                ; do silly things
17:                (let loop ()
18:                  (print "Agent alive")
19:                  (ulm:pause)
20:                  (loop))))))
21:       ; move the agent and its side-kick
22:       (ulm:migrate-to "host" name))))
```

Here we have a thread created by the implicit thread (line *1*), which has the local site as parent. This thread creates an agent (line *6*) and then migrates it via objective migration (line *22*). This agent will only have its first instant executed on the local site, prior to migration. During that first instant, it creates a second thread (line *10*), and then prints in a loop (lines *17-20*). That second thread will have the agent as parent, and so will migrate with him at the end of the instant. Just like its parent, this thread will only execute locally during its first instant, prior to migration, and so the execution on the local site will be such:

```
Our parent is the local site
Agent alive
Second thread
```

After that, both the agent and its child thread will resume execution and printing on the remote site host. Migration groups are used to keep consistency between an agent and the threads it needs to function. It has the additional benefit that these groups keep a local non-blocking access to the agent's references throughout migrations.

3.3 Confinement of Non-Determinism

While the execution of threads and agents within a given instant is entirely deterministic, the physical migration of agents between sites is intrinsically non-deterministic. This is why agents migrate between instants: it isolates non-determinism between instants, at a well defined place. However, sites need not share a global instant. Rather, each site has its local instants, and when an agent changes site, he leaves a local instant to enter another on the new site. Even when sites exchange agents, the number of local instants on each site that lapse during the physical transportation of the agents is arbitrary and non-deterministic. In particular, if an agent A leaves the site S_1 for the site S_2 which is locally at instant I_i (at the time when the agent leaves), the agent can arrive at the instant I_{i+1+n} on S_2, with $n \geq 0$. The inter-instant migration of the agent can be seen as a first local inter-instant phase, and a later second inter-instant phase at the destination site.

3.4 Shared Variables

The behaviour of the first example of migration is quite simple, but migration introduces questions regarding variables that are shared between migrating agents and the threads that stay behind. What happens when two threads sharing a variable are separated is a classical question among mobile languages [2,14]. Let us illustrate one of those problems:

```
1: (let ((shared-var 2))
2:   (ulm:agent
3:     (lambda ()
4:       (ulm:migrate-to "other-host")
5:       (ulm:pause)
6:       ;; we are now on 'other-host'
7:       (set! shared-var 5)
8:       (print shared-var)))
9:   (ulm:pause)
10:  (set! shared-var 9))
```

In this example we create an agent that migrates to other-host to set the variable shared-var and print it (lines *7* and *8*).

Here we have the local variable shared-var, which is free in agent's body: it is allocated in the calling thread's stack. When the agent migrates and keeps using this variable we have a problem: this local variable is being used by two different threads on

different hosts.

Some other languages transform any free variable in an agent's body into a remote *proxy*. Proxies are a way to reference variables across the network in a transparent manner, reading or setting it through the proxy causing network communication between the proxy and the remote variable. The remote references in ULM do not have transparent proxy semantics though. Besides, ULM references are explicitly declared, accessed and read (see Section 3.6), so they are in no way transparent. Furthermore, turning these free variables into references would block the program as soon as line *7*, which is not what one would expect.

Using proxies here would solve this problem, but to what cost? ULM references were created to offer a reliable deterministic semantics of execution locally. This is what GALS means: communications across sites are unreliable and accessing a local variable should in no way introduce non-deterministic behaviour in a thread.

3.5 Migration by Copy

In order to solve the *free variable* problem, we have decided to migrate them by copy. Here is an example to illustrate the copying of free local variables:

```
 1: (define (remote-run remote-host thunk)
 2:   (let ((set-signal (ulm:signal))
 3:         (val 'undef))
 4:     (ulm:agent
 5:       (lambda (name)
 6:         (ulm:migrate-to remote-host)
 7:         (ulm:pause)
 8:         ; we're now on remote-host
 9:         (set! val2 (thunk2))
10:         (ulm:migrate-to "home")
11:         (ulm:pause)
12:         ; we're now back home
13:         (ulm:emit set-signal)))
14:     ; wait for the agent to return
15:     (ulm:await set-signal)
16:     ; return the value
17:     val))
```

This is a first attempt at implementing an RPC (Remote Procedure Call), which unfortunately does not work as intended, as will be explained below. The procedure `remote-run` takes a `remote-host` and a `thunk` as parameters and sends an agent on `remote-host` to execute that `thunk` and return its value. The caller thread waits for the agent to come back by waiting on a shared signal (line *15*), which will be emitted by the agent when it returns (line *13*). Note that executing an agent's body does not yield any value, since it can terminate anywhere and would not know whom to return that value to.

This (wrong) example allocates a `val` variable outside the agent's body (line *3*), which is shared by the agent and the caller thread, but only up to the point when the agent migrates (line *7*). During migration, all free variables used by the agent (underlined in the example) are duplicated for the agent to go along with (marked with $_2$), together with the values associated with those variables at the instant of migration. This is migration by copy. Once the agent comes back home, it still has its own copy of the variable `val` (i.e. $\texttt{val}_2$), which is not the same as the `val` it left behind. Therefore, setting it has no effect for the waiting thread, which will always return an undefined value. We will explain in Section 3.9 why `set-signal` does not suffer from duplication.

With this example, we notice that migration by copy does solve the free variable problem, but is not enough to allow interaction between two threads that have been separated by migration.

3.6 References

This is where ULM references show their value. Let us attempt to solve the last problem with references:

```
 1: (define (remote-run remote-host thunk)
 2:   (let ((ref (ulm:ref))
 3:         (set-signal (ulm:signal)))
 4:     (ulm:agent
 5:       (lambda ()
 6:         (ulm:migrate-to remote-host)
 7:         (ulm:pause)
 8:         ; we're now on remote-host
 9:         (let ((val (thunk)))
10:           (ulm:migrate-to "home")
11:           (ulm:pause)
12:           ; we're now back home
13:           (ulm:ref-set! ref val)
14:           (ulm:emit set-signal))))
15:     ; wait for the agent to return
16:     (ulm:await set-signal)
17:     ; return the value
18:     (ulm:unref ref)))
19:
20: ; go fetch the uptime of "other-site"
21: (remote-run "other-site"
22:   (lambda () (getuptime)))
```

This procedure creates a reference stored on the local site's heap (`(ulm:ref)` creates a new reference, line *2*). It then sends an agent on the `remote-host` (line *6*) to execute the `thunk` there (line *9*) and return (line *10*) with its return value in `val`. Once back, it sets the reference to that value (`(ulm:set-ref! ref var)` affects `val` to the ULM reference `ref`, line *13*), wakes up the caller thread (lines *14* and *16*), which uses that reference to return its value (`(ulm:unref ref)` returns the value of the reference `ref`, line *18*). This example does not illustrate the use of remote references with its blocking semantics, but it does show how references are used by threads separated by migration to share a variable.

In this `remote-run` example, we create a reference to a variable allocated in the local site's heap (line *2*), and the agent migrates with it. During migration, it mutates from a local reference to a remote reference: it becomes a unique distant reference on the other site. But setting it there would block the agent (remember: remote access to references is blocking). Instead, we create a local variable to store the return value of `thunk` (line *9*), and migrate back. Once the agent arrives on the site where the reference is stored, our remote reference becomes a local reference again. Exactly the same reference that was created before leaving, and the same that the caller thread (that stayed here all along) is using. Setting this reference to `val` (line *13*) allows the agent to communicate a value to the caller thread that was waiting for it.

3.7 Global Variables and Modules

Although it may not seem directly relevant to our discussion on mobility, the module system of ULM is presented here because it introduces the definition and scope of global variables. In our implementation of ULM, a global variable is associated to the *module* that declares it. Each module has a list of global variables that can

be *exported* to other modules that *import* them, and a toplevel execution. Here is an example of ULM module:

```
1: (module foo
2:   (import std-scheme ulm)
3:   (export
4:     bar
5:     (gee x)
6:     ))
7:
8: (define bar 2)
9:
10: (define (gee x)
11:   x)
12:
13: (define (mine)
14:   (print "non-exported global"))
```

This declares a module named `foo`, which uses global variables exported by the *ulm* and *std-scheme* modules, and exports the two global variables `bar` and `gee`. Global variables representing closures are exported in a syntax that explicits their prototype (here `(gee x)` to warn importers of `gee` that it is a procedure and takes one argument). Here `mine` is a non-exported global variable containing a closure. It is local to this module and cannot be used by other modules. The exported variables `bar` and `gee` on the other hand can be imported and used by any other module, if they import the module `foo`.

There are two main standard modules in ULM: *std-scheme* which exports some standard Scheme procedures (such as `for-each` or `assq`)[3], and *ulm* which declares and exports every ULM primitive and derived constructs, prefixed with the `ulm:` namespace for clarity.

3.8 Ubiquitous Variables

We now know how to share variables across the network, and migrate with free variables that will be duplicated. What about standard libraries? In the last example, we only talked about free *local* variables, but there are more variables that become free during migration: global variables suffer the same problem. Although you could expect global variables from the current module to be duplicated during migration (just like local variables), variables from other modules (such as our *ulm* or *std-scheme* modules) deserve another treatment: otherwise, migrating any agent would result in migrating copies of whole libraries, which is a waste of bandwidth.

In addition to the local and global variables which are duplicated during migration, there is a type of variable called *ubiquitous*. Ubiquitous variables constitute a category of global variables that are not duplicated during migration. Instead, they are bound dynamically upon arrival on the new site. This imposes a few restrictions though, the first one being that it must be possible to find these variables upon arrival (local variables fall out of this category).

In our implementation of ULM, global variables are bound to modules, and can be exported outside of these modules. Only modules can be declared ubiquitous (using the module declaration `ubiquitous-module` instead of `module` as seen in the last section), which makes all the global variables it exports also ubiquitous. Those modules are called this way because the programmer assumes they can be found everywhere at the same time, that is on every site. Using those ubiquitous variables while migrating means that upon arrival on the new site, they will be dynamically bound to their local counterparts.

Ubiquitous variables allows ULM programs to interact with sites and agents after migration. It is used among other things to move without dragging along whole libraries (such as the standard ones), to be able to call local procedures (like `gethostname`), and to interact with the site or other agents via those local procedures.

Since we declared our *std-scheme* and *ulm* modules ubiquitous, here is what the `remote-run` example looks like with ubiquitous variables underwaved and duplicated variables underlined:

```
1: (define (remote-run remote-host thunk)
2:   (let ((ref (ulm:ref))
3:         (set-signal (ulm:signal)))
4:     (ulm:agent
5:       (lambda ()
6:         (ulm:migrate-to remote-host)
7:         (ulm:pause)
8:         ; we're now on remote-host
9:         (let ((val (thunk2)))
10:           (ulm:migrate-to "home")
11:           (ulm:pause)
12:           ; we're now back home
13:           (ulm:ref-set! ref2 val)
14:           (ulm:emit set-signal2))))
15:     ; wait for the agent to return
16:     (ulm:await set-signal)
17:     ; return the value
18:     (ulm:unref ref)))
19:
20: ; go fetch the uptime of "other-site"
21: (remote-run "other-site"
22:   (lambda () (getuptime)))
23:   )
```

Here it is clear that all the *`ulm:...`* procedures are ubiquitous, as is `getuptime`, since we want to get the uptime of the site to which we migrate. All local variables are duplicated by the migration, as is the global variable `remote-run`, which is not ubiquitous[4].

3.9 Special Values

You will notice that references and signals are underdashed instead of underlined in the previous example. This is because they are in effect duplicated during migration, but their values are special. We already explained that references can change state (local/remote) during migration but remain unique on each site: this is why `ref` is the same as $\texttt{ref}_2$ upon return of the agent (lines *13* and *18* in the last example).

Signal values are also special: they are associated a universal value, which will always be equal after migration, in the same way strings that are not `eq?` can be `equal?`. This explains why emitting $\texttt{set-signal}_2$ (line *14*) awakes the thread waiting on `set-signal` (line *16*).

3.10 How It All Fits Together

We have described informally the behaviour of free local variables, global variables, ubiquitous global variables, and two special kinds

[3] Our ULM Scheme is not R5RS compliant.

[4] Actually, `remote-run` is never used by the agent after migration, so it is not necessary to duplicate it.

of values with respect to migration. It should be noted that aside from the signals and references values, only variables are concerned by migration. In particular, values such as pairs or vectors, which can contain other values, do not act like variables and the distinction between ubiquity or copy is never relevant to values. To illustrate the distinction, two agents cannot share a value through a pair's content unless that pair has been obtained by a common variable since their last migration, and if both agents are on the same site.

Our experience in programming with ULM is that the distinction between these types of variables or values is quite intuitive. Anything you want to keep sharing after migration has to be explicitly declared (through references). The other variables will be taken care of: that is, whether they are dynamic or duplicated, your program will keep running after migration. Suspension on a reference is explicit (via *ulm:ref-set!* for example), so the programmer knows where potential suspension happens.

Whether the variables are dynamic or duplicated is left to the module designer that provides the variable (in the case of global variables), so, for instance the programmer does not need to know (in most cases) whether his implementation of `map` comes from one site or another. In any case, the module designer knows what to declare ubiquitous.

The only thing that might surprise programmers at first is the free variable duplication, if they try to use them as a communication means between migrated agents and sites for example. But this is a habit worth losing in the case of ULM because inter-agent communication can be done via references, signals or dynamic variables. The philosophy behind these types of variables is that local intra-agent execution is the default. Any inter-agent communication is explicitly marked so.

4 Extended Example

We now present an example which illustrates the benefits of ubiquitous variables, along with a mobile reactive chase. The following is a prey/predator example in which rabbits try to escape a fox. Rabbits eat grass in a field (we will use a field per site) until they are fed up or hear a fox arriving or killing another rabbit, in which case they migrate to a random site and leave a trail. The fox makes noise when he arrives in a site, and then hides until rabbits come in or he gives up. When a rabbit comes in the fox kills the first one and goes away. In this example, ubiquitous variables (except those from the *ulm* and *std-scheme* modules) are underwaved:

```
 1: (ubiquitous-module salad-field
 2:   (import std-scheme ulm)
 3:   (export
 4:     kill         ; local signal
 5:     fox-arriving ; local signal
 6:     eat-grass    ; local signal
 7:     (go-away)
 8:     (follow-trail)
 9:     ))
10:
11: (define *trails* '())
12:
13: (define kill (ulm:signal))
14: (define fox-arriving (ulm:signal))
15: (define eat-grass (ulm:signal))
16: (define (go-away)
17:   (let ((dest (random-other-site)))
18:     (set! *trails* (cons dest *trails*))
19:     (ulm:migrate-to dest)
20:     (ulm:pause)))
21:
22: (define (follow-trail)
23:   (let ((dest (if (pair? *trails*)
24:                 (car *trails*)
25:                 (random-other-site))))
26:     (ulm:migrate-to dest)
27:     (ulm:pause)))
```

We define an ubiquitous ULM module with three signals and two procedures exported. These exported variables represent signals and procedures local to a site: in this case a signal to indicate the fox's arrival (`fox-arriving`), one emitted by rabbits while eating (`eat-grass`), and another to represent the fox killing a rabbit (`kill`).

We now define a non-ubiquitous module where the behaviour of rabbits and foxes are each defined in a procedure:

```
 1: (module fox-rabbit
 2:   (import std-scheme ulm salad-field))
 3:
 4: (define (rabbit name)
 5:  (let ((killme (ulm:signal)))
 6:   (let watchout-loop ()
 7:    (ulm:watch killme
 8:      (lambda ()
 9:       (let ((bored #f))
10:        (print name " Rabbit Arriving")
11:        (ulm:watch-or (list fox-arriving kill)
12:          (lambda ()
13:           (let eat-loop ()
14:            (ulm:emit eat-grass killme)
15:            (print name " Rabbit Eating")
16:            (ulm:pause)
17:            (if (= 1 (random 5))
18:             (set! bored #t)
19:             (eat-loop)))))
20:        (if bored
21:         (print name " Rabbit Bored")
22:         (print name " Rabbit Fleeing"))
23:        (go-away)
24:        (watchout-loop)))))
25:   ; we got killed
26:   (print name " Rabbit Dead")))
```

The behaviour of the rabbit agent is defined in the `rabbit` procedure. The rabbit starts by creating a signal (line *5*) by which it will be identified in case it gets killed. It then enters a preemption block on that signal (line *7*): when that signal is emitted, the rabbit dies. In that block it is going to eat (line *13*) for a random amount of instants while watching out for a fox arriving or the fox killing another rabbit (this is a variant of *ulm:watch* which preempts on any presence in a set of signals, line *11*). The eating consists in emitting the `eat-signal` with the signal representing the rabbit's life as value[5] (line *14*). When the rabbit is bored (line *18*) or is preempted by the fox's arrival or killing another rabbit (line *11*), it goes away and loops (lines *23* and *24*).

Notice how `kill`, `fox-arriving`, `eat-grass` and `go-away` are ubiquitous variables: they are dynamic per-site.

[5]This introduces valued signals: you can assign several values to a signal during the instant, the first one of which sets it as emitted.

The fox's behaviour is defined in the procedure that follows:

```
 1: (define (fox name)
 2:  (let loop ()
 3:   ; arrive somewhat noisily
 4:   (print name " Fox Arriving")
 5:   (ulm:emit fox-arriving)
 6:   (ulm:pause)
 7:   ; wait for a rabbit silently
 8:   (let wait ((i 20))
 9:    (if (> i 0)
10:     (let ((eaters (ulm:present eat-grass)))
11:      (if eaters
12:       (begin
13:        (print name " Fox Killing "
14:                    (car eaters))
15:        (ulm:emit kill)
16:        (ulm:emit (car eaters)))
17:       (begin
18:        (print name " Fox Hiding")
19:        (wait (- i 1)))))))
20:   (print name " Fox Going")
21:   ; follow the first trail
22:   (follow-trail)
23:   (loop)))
```

The procedure *(ulm:present s)* (line *10*) is used to query the presence of the signal `s` at the current instant. If `s` is emitted during this instant, *ulm:present* will unblock at the current instant and return any value associated with it when it was emitted. But there is no way to know whether a signal will not been emitted within an instant, because we only know its absence when we have decided to stop running threads in an instant. This is called the end of instant, so in ULM, absence can only be determined at the end of instant, and since no thread can run between instants, reaction to absence is always done in the next instant. The behaviour of *ulm:present* when `s` has not been emitted during the current instant is to return `#f` at the next instant.

The fox emits the `fox-arriving` signal in the instant it arrives (line *5*), then it hides and waits for the first broadcast of `eat-grass` (line *10*), which is emitted by any eating rabbit. If there is nothing, that call is blocking (implicit cooperation) and we can safely loop because we're already at the next instant when it returns (line *19*). If there is a rabbit, *ulm:present* returns the list of `killme` signals emitted by each rabbit while eating. Each signal represents the life of a rabbit (the outer `watch` block of his loop, line *7* of the rabbit procedure). The fox can then emit the `kill` signal to warn all rabbits (line *15*), and the signal that will kill the first rabbit that showed up (line *16*). After that, the fox follows the first trail it finds and goes on another site (line *22*).

Note that the killed rabbit is preempted twice in the same instant: once by the emission of its `killme` signal, and the second time by the `kill` signal. Whenever several watch blocks should be preempted, it is the outermost block that is preempted, thus killing the rabbit without making him flee. It is also worth noting that since the preemption is weak, the rabbit still has time to chew his last bit of grass before dying[6].

The use of ubiquitous signals and methods to represent what happens in each site enables us to start rabbits and foxes on any different site, and still have them interact on each site according to the local signals and procedures.

[6]Which clearly shows the total lack of resemblance between these rabbits and Evil ones with Big Sharp Pointy Teeth.

5 Implementation

We chose Scheme as the host language for the ULM primitives in order to concentrate on the reactive and mobility issues and not on complex host language syntax or semantics. Due to the strong migration semantics, we opted for a bytecode compiler/VM couple, to avoid having an interpreter stack while executing ULM programs, since stacks are typically not first-class objects (and indeed not in the target executables our interpreter is compiled in: Java, C, .NET). We use an academic bytecode interpreter from Queinnec [3] to compile scheme primitives to bytecodes and to execute those bytecodes. We also use the syntactic macros from Bigloo [13] along with its implementation of most of the standard Scheme library used from ULM. The compiler and VM are also written in Bigloo Scheme, for portability and native execution.

5.1 Reactivity

Reactive ULM primitives and scheduling are managed in the *ulm* module, implemented in the host language itself, with only three VM primitives added. In the VM, each thread is represented by a closure object, a stack and any register that needs saving by the VM when changing context (such as the program counter, stack pointer, etc...). In the *ulm* module, scheduling is done with an extra thread that is executed only at the end of instants to initiate the next instant. The scheduling of threads during the instant is done by the threads themselves, whenever they emit or wait for signals. In theory the scheduler thread is optional since the *end of instant* could be executed in any other thread, but using an extra thread made things much easier to write and understand.

5.1.1 Contexts

Suspension and preemption are represented by a list of *WW-cells* (When/Watch cells) that are augmented with either a When-cell or a Watch-cell when entering a suspension or preemption block (resp.). When-cells specify the signal of the *ulm:when* block, and a boolean that indicates whether it is satisfied for this instant (it is satisfied if the signal has been emitted). Watch-cells associate the preemption signal with a procedure that can escape from the *ulm:watch* block. Preemption is implemented using `bind-exit`[7]: each time a *ulm:watch* block is entered, we enter a `bind-exit` block and associate its `exit` procedure with the preemption signal.

5.1.2 End of Instant

At the end of the instant, the scheduler thread walks the list of threads and reverts all When-cells to `'unsatisfied`. Then for the outermost Watch-cell that is satisfied, the scheduler notifies the thread that it should be preempted (we will see where this is done later). Any thread with no `'unsatisfied` When-cells (that includes preempted threads) is scheduled to run at the next instant.

5.1.3 Scheduling

We already revealed that intra-instant scheduling is done by the threads themselves, during various ULM primitive calls. Emitting a signal causes the list of threads waiting for it to be examined. Each unsatisfied When-cell is checked and threads that only have satisfied When-cells are rescheduled for later in the instant. In other words, threads that have several When-cells are only allowed to run

[7]A form of `call/cc` whose *escape* procedure is only valid in its dynamic extent.

when all have been satisfied. This enables us to have threads waiting for *n* signals only be present in any one signal queue at a time. A thread will thus hop from one waiting queue to the other (as each queue is satisfied) in the worst case, but be considered for scheduling only once in the best case (if it is in the queue of the its last unsatisfied signal).

5.1.4 Context Switching

Context switching is done when waiting for a signal that hasn't been emitted yet (by entering a *ulm:when* block[8]). When that happens, the thread blocks and finds the next thread to schedule and tells the VM to switch its context to it. Here is the procedure that switches context in the *ulm* module:

```
 1: (define (cooperate)
 2:   ; tell the VM to switch context
 3:   ; to the next thread
 4:   (switch-to-thread (get-next-thread))
 5:   ; treat preemption
 6:   (if (thread-preempted? *current-thread*)
 7:     ; get the Watch-cell of the signal
 8:     ; that preempted us
 9:     (let* ((watch-cell (thread-preempted-cell
10:                         *current-thread*))
11:            (exit (cddr watch-cell)))
12:       (exit))))
```

We can see here that all blocked threads are in the `cooperate` procedure, blocked in the call to the `switch-to-thread` VM primitive. When there is no thread left to schedule, `get-next-thread` returns the scheduler thread, which declares the end of instant.

5.1.5 Preemption

Preemption is decided at the end of the instant, and executed in the next instant. Since all threads unblocked return from the `switch-to-thread` call, we check for preemption there, before returning from `cooperate`. When preemption is needed, we find the `exit` escaper associated with the preempting signal, and execute it, thus unwinding at the end of the `watch` call. This is effectively done in the new instant and ensures that any preemption handlers (such as `unwind-protect`[9]) are called in the new instant and not during the end of instant phase.

5.2 Mobility

Mobility is implemented mostly in the VM, because serialisation of the thread state (stack, memory and bytecodes) requires extensive access to data that should be kept away from the interpreted language. Migrating a thread from one site to another consists in finding all accessed (and future accesses to) variables and bytecodes, modification of bytecodes, serialisation, transport, deserialisation and integration.

5.2.1 Finding Accessible Variables

Finding all accessible variables is done by looking through the current environment, the stack, and the bytecode of all accessible closures. Each accessible object is assigned a unique serial number used later to resolve circular references. Ubiquitous variables are not traversed, since they will be dynamically bound after migration.

5.2.2 Modification of Bytecodes

The bytecode needs to be modified before migration for two reasons: first, the migrating agent will become a special kind of module during migration, and second because this is where variable duplication takes place. Making a module out of each migrating agent allows us to simplify the encapsulating process because an agent migrates with bytecode, a list of constants, a list of global variables and a name, which fits perfectly the role of modules and makes inserting an agent in a site fairly easy. These are special modules however, in the sense that they do not export any variable and cannot be imported. The bytecode modification is needed because during the search for accessible variables, we come across global variables that are not ubiquitous and need special treatment.

There are three types of bytecodes to access variables: `LOCAL-REF/SET`, `GLOBAL-REF` and `IMPORTED-REF`. `LOCAL-REF/SET` represent local variables, that is, lambda parameters, and they can never be ubiquitous. `GLOBAL-REF` and `IMPORTED-REF` are both global variables but the first one is a global variable from the current module and is indexed, while the second one is an imported global variable, referenced by module and global names.

`GLOBAL-REF` bytecodes need to be changed into `IMPORTED-REF` if the current module is ubiquitous, or *phagocytized* otherwise. We call phagocytizing a global variable the action of duplicating it, and adding it to the module we create for the agent migration. In effect the agent keeps using that global variable, but it is relocated in its own module, changes index and migrates with a copy of its current value.

In a similar way, `IMPORTED-REF` bytecodes that refer to non-ubiquitous variables need to be phagocytized and changed into a `GLOBAL-REF`. The closures that are not referenced through ubiquitous variables also need to be phagocytized, since we need to add their bytecode to the agent's module, bytecode which has to be relocated and modified.

5.2.3 Serialisation and Transport

Serialisation is done by iterating all the values we affected serial numbers to, and using either introspection or specialised treatment to create an *alist* structure to represent them. References are serialised specially, by mutating their state to remote where applicable: all local references not allocated in the agent's heap become remote before migrating. Local references that stay behind but point to the migrating agent's heap also become remote. These serialised values, along with the agent's module and a pointer to the agent's thread structure (which happens to be the root of serialisation) are then sent along the network asynchronously.

5.2.4 Deserialisation

On the other site, an asynchronous thread waits for incoming agents and stores them during the ULM instants until synchronous incorporation at the end of instant phase. Deserialisation is done in two phases: allocation of all the objects that have a serial number, and affectation of all these object's members that were referred to by serial number. Reference mutation also happens during this phase: remote references held by the agent that point to a local heap be-

[8] *ulm:pause* and *ulm:await* are derived from *ulm:when*.

[9] A variant of `dynamic-wind` with no *before* block.

come local, as do remote references present on the site that point to the new agent's heap.

5.2.5 Integration

Integration is the simplest phase, since after deserialisation we are left with a pointer to the agent thread, and its module. We simply load that module, add the agent thread to the list of threads and notify the *ulm* library that there is a new agent to schedule.

5.3 Migration Examples

Now that we have seen all the details of global variables and migration, here is an example of how migration actually works, as far as the programmer is concerned:

```
 1: (module home-mod
 2:  (exports
 3:   home-var
 4:   (home-fun arg)))
 5:
 6: (define home-var 3)
 7:
 8: (define (home-fun arg)
 9:   (set! home-var arg))

10: (ubiquitous-module ubiq-mod
11:  (import home-mod ulm)
12:  (exports
13:   ubiq-var
14:   (ubiq-fun a b)))
15:
16: (define ubiq-var "dynamic")
17:
18: (define (ubiq-fun a b)
19:   (* a b))
20:
21: (ulm:agent
22:  (lambda ()
23:    (ulm:migrate-to "host")
24:    (ulm:pause)
25:    (print ubiq-var)
26:    (ubiq-fun home-var 2)))

27: (module main-mod
28:  (import home-mod ubiq-mod ulm))
29:
30: (define my-var "hello")
31:
32: (ulm:agent
33:   (lambda ()
34:     (home-fun 5)
35:     (ulm:migrate-to "host")
36:     (ulm:pause)
37:     (print my-var)
38:     (ubiq-fun 2 3)))
```

We define three modules: `home-mod` exports non-ubiquitous global variables, `ubiq-mod` defines and exports ubiquitous global variables and sends an agent to `host`, while `main-mod` defines non-ubiquitous variables and also sends an agent to `host`. After the transformations of bytecode during migration, here is how the two agents would look like upon arrival on *host* if their bytecode was disassembled into this fictitious code:

```
 1: (agent-module agent1
 2:  (import ubiq-mod ulm))
 3:
 4: (define home-var 5)
 5:
 6: (define __agent1-body
 7:  (lambda ()
 8:    (ulm:migrate-to "host")
 9:    (ulm:pause)
10:    (print ubiq-var)
11:    (ubiq-fun home-var 2)))

12: (agent-module agent2
13:  (import ubiq-mod ulm))
14:
15: (define home-var 5)
16:
17: (define (home-fun arg)
18:   (set! home-var arg))
19:
20: (define my-var "hello")
21:
22: (define __agent2-body
23:   (lambda ()
24:     (home-fun 5)
25:     (ulm:migrate-to "host")
26:     (ulm:pause)
27:     (print my-var)
28:     (ubiq-fun 2 3)))
```

This example illustrates several things. First, that the agents become their own module (illustrated by the fictitious `agent-module` directive). To their modules are added copies of any non-ubiquitous global variable they were using (from their module or any other): what we call phagocytizing. All ubiquitous global variables (like `ulm:pause` or `ubiq-fun`) mutate (if not already) into a dynamic variable bound upon arrival to the new site's equivalent variables.

Note that while agent modules are a practical implementation technique, they are not directly available to the programmer, who will likely never need to know about them.

6 Food For Thought

The ULM interpreter we have implemented is a prototype: it implements the semantics of Scheme and the ULM primitives, and adds the notion of ubiquitous variables for the migration semantics. However, there are a number of things that still need to be worked on: better compilation analysis and optimisation and a distributed garbage collector. The possibility to call native code from ULM and have that native code call back ULM code has been implemented, and we are studying its implications regarding mobility.

Some other enhancements would benefit directly to the ULM primitives: during compilation, bytecodes for global variable access could be adapted for ubiquitous modules (this is currently done during migration), as long as it does not impede on non-mobile execution (since we expect migration to be less frequent than global variable access). Better code analysis and compilation could lead to reduced memory traversal during migration (unused variables or dead code for example).

Implementation of derived ULM procedures (such as `ulm:pause` or `ulm:present`) would benefit from direct support in the reactive engine, instead of being implemented via ULM primitives.

An object system in ULM could allow us to implement classical distributed proxies that could be used in migration to implement dif-

ferent argument passing styles for RPC (copy, migrate, visit, lazy, etc...). Representing signals as objects could also open new ways to interact with them. A preliminary work on implementing a mixin object system as defined by G. Boudol [9] has been done, but is not presented in this paper.

Mechanisms in case of migration transport failure or node unavailability have not been studied yet, as these types of failures are hard to represent semantically. We could imagine that migration would return a signal that would be emitted in case of successful arrival, with a notion of timeouts and retries (any agent that can migrate can also be saved on disk[10] to be used for retries or reentry).

The notion of ubiquitous modules implies that an agent expects them to be on every site it visits. Mechanisms in case of missing ubiquitous module upon arrival have not been studied. Automatic retrieval, migration failure or blocking the agent are possible answers, although the last one fits best the philosophy behind ULM. Reifying modules as first-class objects could also provide clues on how to treat this, as agents could perhaps fetch and load modules explicitly during execution.

Although the ULM primitives integrate well with `bind-exit` and `unwind-protect` (both presented in Section 5.1.1), this is because `bind-exit` can be seen as an intra-instant preemption akin to *`ulm:watch`*. `call/cc` on the other hand, interferes with ULM primitives, and introduces questions regarding the passing of continuations between threads which we do not wish to allow.

Even though an agent can acquire new procedure values during migration, which means the agent will collect the bytecode, together with the captured and global variables of that procedure, each site's GC ensures that they will be collected when not used anymore. In any case, the traversal done prior to migration will not collect unused procedures, and dead code analysis will help in leaving behind any variable access that will never be reached in the bytecode that needs to be taken. There is no reason why agents would grow upon each migration unless they need to by collecting useful procedures. The captured bytecode could however often be shared between agents and even with local sites, but we have not studied such a mechanism for our prototype.

7 Related Work

Few other languages offer both Mobility and Reactivity together with a strong deterministic semantics. Junior [6] offers both a deterministic semantics and reactive programming but only reactive mobility: the state of the reactive engine can be migrated, but not the state of the host language (Java in this case). In Junior, this does not have a big impact, since aside from the reactive instructions, non-reactive Java code is supposed to be atomic and have finished execution between instants. ULM on the other hand allows reactive instructions (including migration) to be called by non-reactive instructions.

ESTEREL [8] is the reactive language from which most reactive primitives in ULM are inspired. Its model of execution is however very different from ULM because it is based on calculation rather than discovery. In our model we discover emitted signals as we execute the code that emits them, while in ESTEREL the presence or absence of signals is calculated for each instant, and holds throughout the entire instant. Because of that, ESTEREL is less modular and dynamic than ULM. It also does not provide mobility.

[10]This is the notion of *checkpoints* in Eden [1] and Emerald [5].

Bigloo FairThreads [6] add a reactive library to Bigloo Scheme (on which parts of our reactive module are based), but do not support migration. On the other hand, FairThreads support multiple schedulers and asynchronous threads (that interact in a deterministic way with synchronous threads by becoming synchronous during interaction).

Concurrent ML [11] offers a preemptive scheduling of multiple threads in a functional language. Communication and synchronisation between threads in CML is done via channels that can be shared by multiple threads, whereas ULM synchronisation is done via broadcast events. It should be possible to program in a similar way to ULM's signals, but critical sections have to be explicitly marked, as with traditional preemptive schedulings. CML does not seem to have a preemption mechanism and does not offer mobility in the language.

Obliq [12] is a functional language that supports strong mobility and remote references through *proxies*. The semantics of their migration is to transform every free variable (here, all variables defined prior to migration become free) into remote references. Obliq also supports threads but the parallel, non-deterministic kind.

Kali Scheme [10] proposes mobile procedures that serve as RPC, with the client providing the server with the procedure it wants it to run. The remote procedure shares memory with the caller via *Address Spaces* and proxies, which is comparable to ULM's heaps and references, except that ULM's remote access is blocking while Kali Scheme is proxied. Thread mobility is done by capturing the current continuation of the thread and executing it remotely. Their migration of the stack reuses a concept found in Emerald [5] where only the top stack frames are migrated, while the rest are migrated on-demand.

Erlang [4] is a functional language aimed at distributed programming. It provides mobility in the same form as Kali Scheme, by spawning an asynchronous process in a remote host by sending a procedure there. Communication and synchronisation between threads (remote or local) is done through messages and queues with unidirectional and synchronous communication. The scheduling of Erlang threads is preemptive, so critical sections have to be explicitly marked, and the execution of non-perfect critical sections is non-deterministic and suffers from the usual debugging problems.

8 Conclusion

In this paper we have presented the ULM primitives and how to use them to implement multi-threading and mobile programs that interact with other unknown programs. We have shown the questions mobility poses regarding variable access, and have proposed different solutions to address them while preserving the local execution reliability that ULM offers. We introduce duplicated and dynamic variables for the Scheme implantation of the ULM primitives we implemented, and show how this is intuitive for the programmer. At the same time, we show how to use dynamic variables to interact with other unknown agents.

Our prototype implementation of the compiler and virtual machine serves as proof-of-concept for the ULM specification, and enables us to implement functionalities as complex as RPCs with very few lines of code. Indeed, we believe the set of primitives ULM provides is powerful enough to implement different types of distributed programming techniques, while providing a clear and predictable framework.

9 Bibliography

[1] A.P. Black – **The Eden Programming Language** – Technical Report 85-09-01, Dept. of Computer Science, University of Washington, Seattle, Washington, September, 1985.

[2] Alfonso Fuggetta and Gian Pietro Picco and Giovanni Vigna – **Understanding Code Mobility** – IEEE Trans. Softw. Eng., 24, (5), 1998, pp. 342–361.

[3] Christian Queinnec – **Lisp in Small Pieces** – *Cambridge University Press*, 1996.

[4] ERLANG – **http://www.erlang.org**.

[5] Eric Jul and Henry Levy and Norman Hutchinson and Andrew Black – **Fine-Grained Mobility in the Emerald System** – ACM Transactions on Computer Systems, 6, (1), New York, NY, USA, February, 1988, pp. 109–133.

[6] FairThreads – **http://www-sop.inria.fr/mimosa/rp/FairThreads** – *MIMOSA - INRIA*.

[7] G. Boudol – **ULM: a core programming model for global computing** – Proceedings of ESOP 04, Lecture Notes in Computer Science (LNCS), 2004, pp. 234-248.

[8] Gerard Berry – **Constructive Semantics of Esterel: From Theory to Practice (Abstract)** – Algebraic Methodology and Software Technology, 1996, pp. 225.

[9] Gérard Boudol – **The Recursive Record Semantics of Objects Revisited** – Proceedings of the 10th European Symposium on Programming Languages and Systems, 2001, pp. 269–283.

[10] Henry Cejtin and Suresh Jagannathan and Richard Kelsey – **Higher-Order Distributed Objects** – ACM Transactions on Programming Languages and Systems, 17, (5), September, 1995, pp. 704–739.

[11] John H. Reppy – **Concurrent Programming in ML** – *Cambridge Univ Press*, 1999.

[12] Luca Cardelli – **A language with distributed scope** – Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 286–297.

[13] Manuel Serrano and Pierre Weis – **Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages** – Static Analysis Symposium, 1995, pp. 366-381.

[14] Tatsurou Sekiguchi and Akinori Yonezawa – **A calculus with code mobility** – Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems, 1997, pp. 21–36.