The mobile ambient calculus in Common Lisp

Joshua Taylor
tayloj@cs.rpi.edu

December 14, 2009

Abstract

This document describes an implementation of the mobile ambient calculus in Common Lisp. The current implementation achieves non-deterministic interleaving of processes and provides tracing capabilities, but execution is not genuinely concurrent. The programming interface is designed to imitate the ambient calculus closely, allowing a straightforward translation of existing ambient calculus expressions, but abstraction is also available via Common Lisp's strong macro facility, and processes are provided that can use Common Lisp code directly. As examples of extensibility, true objective movements and safe ambients are implemented. Familiarity with both Common Lisp and the ambient calculus is presumed, though some exposition on each is given where appropriate.

Contents

1 Introduction

1.1 Getting Started

The system is available at http://www.cs.rpi.edu/~tayloj/mobile-ambients/ along with full documentation. The library has been tested with LispWorks 5.1.2, Steel Bank Common Lisp 1.0.30, Clozure Common Lisp 1.3, and Allegro 8.1. Once the library (just `mobile-ambients.lisp`) is downloaded, it can be compiled and loaded. In an interactive session, this might look like the following. (In transcripts of interactive sessions, `?` is the Lisp's prompt.)

```
? (compile-file "mobile-ambients")
#P".../mobile-ambients.dx64fsl"
NIL
NIL
? (load "mobile-ambients")
#P".../mobile-ambients.dx64fsl"
```

The library defines a package named `mobile-ambients`, and the nickname `amb` is defined as well, so symbols in the `mobile-ambients` package, e.g., `process` can be referenced as `amb:process`. In this document, we do not want to type `amb:` before every symbol in the `mobile-ambients` package, so we begin by moving the interactive session into the `mobile-ambients-user` package.

```
? (in-package #:mobile-ambients-user)
#<Package "MOBILE-AMBIENTS-USER">
```

If everything has worked until now, the library is probably loaded properly, but we can try running a process to be sure. Let us try running the mobile ambient process $n[] \mid m[in\ n.P]$. With any luck, we will see that ambient m is able to move into ambient n.

```
? (run-process (par (amb 'n) (amb 'm (in 'n (named 'P)))))
; *. N[] | M[in N.P]
; 0. N[] | M[in N.P]
; 1. N[M[P]]
#<N[M[P]]>
```

Now that everything is up and running, we can look into how the system works.

2 A Generic Process Engine

In §1.1 we saw a simple example in which `run-process` was applied to `(par (amb 'n) (amb 'm (in 'n (named 'P))))`. The latter describes the process $n[] \mid m[in\ n.P]$ and we will see how processes are constructed in §3. For the moment, we consider `run-process`.

`run-process` takes a process to run, and runs it. Specifically, `run-process` invokes the generic function `evolutions` on the process to produce a list of possible evolutions. Each evolution is a function of no arguments that, when invoked, will modify the process, evolving it in some way. If `evolutions` returns no processes, `run-process` returns immediately. Otherwise, `run-process` randomly selects one of the evolutions and invokes it. `run-process` continues to get evolutions for the process and invoke them until there no more evolutions are produced. `run-process` takes a second optional argument, a boolean, and if it is true, `run-process` will print the process after each evolution is invoked. (The default for the second argument is true. This is why we saw a trace in the example in §1.1.)

`run-process` actually does a little bit more than what we just described. Before calling `evolutions` on the process the first time, and after invoking each evolution, `run-process` calls the generic function `cleanup` on the process. `cleanup` methods are designed to perform evolutions that clean up a process. For instance, a `cleanup` method for parallel compositions removes any instance of **0**, and 'lifts' any nested parallel compositions. For instance, `cleanup` turns $P \mid \mathbf{0} \mid (Q \mid R)$ into $P \mid Q \mid R$. The rationale for `cleanup` is that `evolutions` methods can be simpler if they can assume that a process is in some cleanest form, and that these kind of transformations are so desirable that they should happen automatically, without having to wait to be randomly selected as evolutions.

This simple `run-process` interface allows any object on which `evolutions` and `cleanup` methods are implemented to be run as a process. Default `cleanup` and `evolution` methods are defined for the `process` class that do nothing but return `nil`.

By defining appropriate `evolutions` and `cleanup` methods, we can call `run-process` with lists. We define a `shorten` function that destructively shortens a list by removing its second element, and a `cleanup` list that does nothing. The `evolutions` method returns a list of single evolution that destructively shortens `evolution`'s argument, if it is long enough to be shortened. Otherwise `evolutions` returns an empty list.

```
? (defun shorten (list)
    (setf (rest list) (rest (rest list))))
SHORTEN

? (defmethod evolutions ((l list))
    (if (endp (rest l)) '()
      (list #'(lambda () (shorten l)))))
#<STANDARD-METHOD EVOLUTIONS NIL (LIST) 21C5FC13>

? (defmethod cleanup ((l list))
    nil)
#<STANDARD-METHOD CLEANUP NIL (LIST) 200E62BF>

? (run-process (list 1 2 3 4 5))
; *. (1 2 3 4 5)
; 0. (1 2 3 4 5)
; 1. (1 3 4 5)
```

```
; 2. (1 4 5)
; 3. (1 5)
; 4. (1)
(1)
```

3 Capabilities and Processes

The previous section showed how the function `run-process` could be used to run anything that can be passed to `evolutions` and `cleanup`. The heart of the mobile ambients library, however, is that `evolutions` and `cleanup` methods have already been defined for a number of entities, particularly those with `process` as a superclass. We now describe the capabilities and processes that implement the mobile ambient calculus.

3.1 Names

In the ambient calculus, ambients are identified by their names. In the library, we use arbitrary Lisp objects as names (though in all of our examples here, we use symbols), and these will be compared with `eql`.

3.2 Capabilities

The minimal mobile ambient calculus has three mobility capabilities. Where *n* is an ambient name, *in n*, *out n*, and *open n* are mobility capabilities. In the library, capabilities are CLOS objects that have `capability` as a superclass. Input, output, and open capabilities are implemented by the classes `in`, `out`, and `open`. These are normal CLOS classes, and can created using `make-instance`. These capabilities all have names, and names can be provided with the `:name` initialization argument to `make-instance`.

```
? (make-instance 'in :name 'n)
#<in N>

? (make-instance 'out :name 34)
#<out 34>
```

This is a tedious way to construct capabilities, and so there are defined constructors, `in^`, `out^`, and `open^`, which take a single argument, a name.

```
? (open^ 'm)
#<open M>
```

3.3 Processes

Processes in the library are also implemented as CLOS classes which include `process` as a superclass. The ambient calculus defines a number of types of process, and so do we, although our collection is slightly different than that given in the formal theory. The `process` class has a `parent` slot which may be read or written with `process-parent` accessor. The parent of a process should be the parallel composition or ambient containing the process. However, a process's parent slot may also be unbound.

3.3.1 Inactive Processes

Inactive processes are implemented by the `inactive` class. These are printed as `0`. They have no evolutions, and `cleanup` does nothing to them (though they may be removed from other processes). Inactive processes can be created with the constructor `zero`, which takes no arguments. Inactive processes are not of much use directly, but can be helpful as default arguments in various places.

```
? (make-instance 'inactive)
#<0>

? (zero)
#<0>
```

3.3.2 Named Processes

Named processes, implemented by the `named` class, are a special kind of inactive process that are displayed by their name. These are useful for debugging and for demonstrations. They are created with `make-instance` with a `:name` initialization argument or by the `named` constructor.

```
? (make-instance 'named :name 'Q)
#<Q>

? (named 'P)
#<P>
```

3.3.3 Compositions

Parallel compositions of processes are implemented by the `composition` class. Compositions may be created with `make-instance` or with the constructor `par`. `make-instance` should be passed a list of processes via the `:processes` initialization argument, and `par` takes an arbitrary number of processes as arguments. When a composition is constructed, it is set as the parent of its composed processes.

```
? (make-instance 'composition
                 :processes (list (named 'P)
                                  (named 'Q)))
#<P | Q>

? (par (named 'P) (named 'Q))
#<P | Q>
```

The `evolutions` method for compositions returns a list containing all of the evolutions of the composed processes, and so computation within a parallel composition proceeds in a non-deterministically interleaved fashion. The `cleanup` method for compositions calls `cleanup` on its composed processes and then lifts composed compositions up to flatten nested compositions (e.g., $P \mid (Q \mid R)$ becomes $P \mid Q \mid R$), and removes inactive processes (i.e., $\mathbf{0}$).

3.3.4 Ambients

Ambients, the heart of the ambient calculus, are parallel compositions of processes identified by names. The `ambient` class includes `composition` as a superclass, and can be created with `make-instance` with `:name` and `:processes` initialization arguments, or with the `amb` constructor, which takes a name and any number of processes. The `evolutions` and `cleanup` methods for `ambients` are the same as those for `compositions`.

```
? (make-instance 'ambient
                 :name 45
                 :processes (list (zero) (named 'R)))
#<45[0 | R]>

? (amb 'n (named 'P) (named 'Q))
#<N[P | Q]>
```

3.3.5 Computations

The library introduces a `computation` process not included in the original ambient calculus. A computation has a function (of no arguments) which can be invoked to perform arbitrary computation. The `evolutions` method for a `computation` returns a list of a single function which will invoke the `computation`'s function. If the result of the function is a process, then the `computation` is replaced by the process, otherwise the `computation` is removed. `computations` can be constructed with `make-instance` and a `:function` initialization argument or by the `fun` macro which takes a sequence of forms and produces the function from them.

For instance, here we define a computation whose function prints (and returns) `P`. Since `P` is not a process, the computation is removed from the composition. Then we define a computation that prints `HELLO` and evolves as $Q \mid R$.

```
? (make-instance 'computation
                 :function #'(lambda ()
                               (print 'P)))
#<{proc}>

? (run-process (par * (named 'Q))) ; * is the previous value
; *. {proc} | Q
; 0. {proc} | Q
P
; 1. Q
#<Q>

? (run-process (par (named 'P)
                    (fun (print 'hello)
                      (par (named 'Q)
                           (named 'R)))))
; *. P | {proc}
; 0. P | {proc}
HELLO
; 1. P | Q | R
#<P | Q | R>
```

The original ambient calculus provided a replication process type, denoted $!P$ which expanded as $P \mid !P$. Replication was used to implement iteration and recursion. Since Common Lisp already supports iteration and recursion, we can create iterative and recursive processes using computation.

```
? (labels ((fact (n &optional (acc 1))
             (if (<= n 1) (print acc)
               (fun (fact (1- n) (* n acc))))))
    (run-process (par (fact 5))))
; *. {proc}
; 0. {proc}
; 1. {proc}
; 2. {proc}
; 3. {proc}
120
; 4.
#<>
```

3.3.6 Actions

Action processes are processes prefixed with capabilities. These block until their capability can be executed, and then proceed as their suffix. Action processes are implemented by the `action` class. They are constructed either with `make-instance` using `:capability` and `:process`

initialization arguments, or by the cap function, which takes a capability and a process, and returns the action process which is the process prefixed by the capability.

```
? (make-instance 'action
                 :capability (in^ 'n)
                 :process (named 'P))
#<in N.P>

? (cap (out^ 'm) (named 'Q))
#<out M.Q>
```

In practice, creating actions with *in*, *out*, and *open* capabilities is common enough that we provide the functions in, out, and open, each of which takes a name and a process and returns the action which is the processes prefixed with the appropriate type of capability with the specified name. (It is because we wanted to reserve these names for action constructors that the capability constructors had the unusual ^ suffix.)

```
? (open 'n (named 'P))
#<open N.P>
```

The evolutions method for actions calls the generic function action-evolutions with the action's capability and the action, and returns its value. (This makes it easier to extend the system with new types of capabilities.) The action-evolutions methods for in, out, and open capabilities produce evolutions that perform ambient entry, exit, and opening.

```
? (run-process
   (par (amb 'n)
        (amb 'm (in 'n (named 'P)))))
; *. N[] | M[in N.P]
; 0. N[] | M[in N.P]
; 1. N[M[P]]
#<N[M[P]]>

? (run-process
   (par (amb 'n (amb 'm (out 'n (named 'P))))))
; *. N[M[out N.P]]
; 0. N[M[out N.P]]
; 1. M[P] | N[]
#<M[P] | N[]>

? (run-process
   (par (amb 'n (named 'P))
        (open 'n (named 'Q))))
; *. N[P] | open N.Q
; 0. N[P] | open N.Q
; 1. P | Q
#<P | Q>
```

3.3.7 Restrictions

The ambient calculus defines a restriction operator $(\nu\ x)$ that is used to introduce fresh names. That is, $(\nu\ x)P$ is a process P in which x is new unique name. Since ambient names in the library are arbitrary Lisp objects, we need no special operator to create fresh names—any new object would suffice.

```
? (let ((new-name (gensym)))
    (amb new-name))
#<#:G5355[]>
```

Nonetheless, it is convenient to have a construct that binds variables to fresh names. The macro `new` does just that, binding variables to fresh symbols whose names are the same as those naming the variables (this makes the printed representation of processes a bit easier to read). Note that symbols with no home package, including named generated by `new`, are printed with a leading `#:`.

```
? (new (x y)
    (par (amb x) (amb y)))
#<#:X[] | #:Y[]>
```

Note that even though these fresh symbols bear a superficial similarity to the symbols that appear in the `new` operator, they are guaranteed to be fresh. Thus, sometimes processes may deceptively appear as though they should evolve when they actually cannot.

```
? (run-process
   (par (new (x y)
          (amb y (in x (named 'P))))
        (new (x)
          (amb x (named 'Q)))))
; *. #:Y[in #:X.P] | #:X[Q]
; 0. #:Y[in #:X.P] | #:X[Q]
#<#:Y[in #:X.P] | #:X[Q]>
```

4 Examples Before Communication

Before describing how the minimal ambient calculus is extended with communication primitives, we will recreate a number of the examples given by Cardelli & Gordon (2000) that use only the minimal calculus. The examples are taken directly from their article (§2.4), as the purpose here is to illustrate how the process calculations can be translated into Common Lisp. It is recommended to read this section along with the original article, if possible. The examples in this section are available online at http://svn.cs.rpi.edu/svn/tayloj/mobile-ambients/examples.lisp.

4.1 Semaphores

Semaphores can be defined as ambients, i.e., a semaphore identified by a name *n* is simply an (empty) ambient named *n*. Acquiring the semaphore is opening the ambient, and to release a semaphore is to introduce an ambient of the specified name. Note that an acquisition of a semaphore should block a process from proceeding until the semaphore is actually acquired. Thus acquisition must be encoded as an action.

```
(defun acquire (n P)
  (open n P))

(defun release (n P)
  (par (amb n) P))

(run-process
 (par (acquire 'n (release 'm (named 'P)))
      (release 'n (acquire 'm (named 'Q)))))
; *. open N.(M[] | P) | (N[] | open M.Q)
; 0. open N.(M[] | P) | N[] | open M.Q
; 1. M[] | P | open M.Q
; 2. P | Q
#<P | Q>
```

4.2 Authentication

In this example, an agent ambient will leave its enclosing home ambient and later return. To continue its inner process upon its return, the agent exposes an ambient with a name originally introduced within the home ambient. This confirms to the home ambient that the agent is trusted (having originated within the home ambient).

```
(run-process
 (par (amb 'home
        (new (n)
          (par (open n)
               (amb 'agent
                 (out 'home
                      (in 'home
                          (amb n
                            (out 'agent
                                 (open 'agent (named 'p)))))))))))
; *. HOME[(open #:N | AGENT[out HOME.in HOME.#:N[out AGENT.open AGENT.P]])]
; 0. HOME[open #:N | AGENT[out HOME.in HOME.#:N[out AGENT.open AGENT.P]]]
; 1. AGENT[in HOME.#:N[out AGENT.open AGENT.P]] | HOME[open #:N]
; 2. HOME[AGENT[#:N[out AGENT.open AGENT.P]] | open #:N]
; 3. HOME[#:N[open AGENT.P] | AGENT[] | open #:N]
```

```
; 4. HOME[open AGENT.P | AGENT[]]
; 5. HOME[P]
#<HOME[P]>
```

4.3 Approximating Objective Moves

In this example, we recreate an approximation of objective moves. (Since the system is extensible, we can implement true objective moves; this is done in §7.1.) Since we lack the fancy harpoon notations in Common Lisp, the abbreviations $n^{\downarrow}$ (an ambient named n allowing entry), $n^{\uparrow}$ (an ambient named n allowing exit), and $n^{\downarrow\uparrow}$ (an ambient named n allowing entry and exit), are called `ambi`, `ambo`, and `ambio`, respectively. In addition, rather than using a distinguished names `enter` and `exit`, these are used as defaults, but can be provided as optional arguments, giving the keyed variants of `ambi`, `ambo`, and `ambio`.

```
(defun allow (n)
  (open n (fun (allow n))))

(defun mv-in (n P &optional (enter 'enter))
  (new (k)
    (amb k (in n (amb enter (out k (open k P)))))))

(defun mv-out (n P &optional (exit 'exit))
  (new (k)
    (amb k (out n (amb exit (out k (open k P)))))))

(defun ambi (n P &optional (enter 'enter))
  (amb n
       (par P)
       (allow enter)))

(defun ambo (n P &optional (exit 'exit))
  (par (amb n P)
       (allow exit)))

(defun ambio (n &optional (P (zero)) (enter 'enter) (exit 'exit))
  (par (amb n (par P (allow enter)))
       (allow exit)))

? (run-process
   (par (mv-in 'n (named 'P))
        (ambio 'n (named 'Q))))
#<N[P | Q | open ENTER.{proc}] | open EXIT.{proc}>
```

5 Communication

The minimal ambient calculus is Turing-complete, but it is much more convenient and reasonable to consider extending the ambient calculus with communication primitives that can be used to transmit values between processes. The ambient calculus defines two more types of processes to achieve this.

5.1 Output

Asynchronous output is achieved with `output` processes. An output process can be constructed with `make-instance` where the value to be output is given by the `:capability` initialization argument (in the ambient calculus, transmitted values are capabilities and ambient names, but we may transmit any Lisp object), or by the `output` function which takes a single argument, the value to be transmitted.

```
? (make-instance 'output :capability 3)
#<{3}>

? (output (out^ 'm))
#<{out M}>
```

The `evolutions` method for output processes examines the surrounding environment of the output process. If there is a sibling `input` process (see the following section), then `evolutions` include an evolution that will transmit the value to the input process and remove the output process. An example of communication is given in the next section.

5.2 Input

Input is implemented by `input` processes. An input process has a function of one argument which is invoked with the received input value when one becomes available and should produce a process, which the input processes evolves as. Input processes can be constructed with `make-instance` and a `:process` initialization argument, which should be a function of one argument, by the `make-input` function which takes a single argument (the function), or by the `input` macro. The `input` macro takes a name a code body, and builds an input process whose function binds the name within the code body.

```
? (make-input
   #'(lambda (x)
       (cap x (named 'P))))
#<().#>

? (input x
    (amb x))
#<().#>
```

The `evolutions` method for input processes always returns the empty list, not because input processes do not evolve, but rather because since input processes only evolve when an output is available, and `evolutions` for the output will produce the necessary evolution. It seems wasteful to have both evolutions computed for both the input and output processes.

```
(run-process
 (par (output (in^ 'n))
      (amb 'n (named 'P))
      (input x
        (amb 'm (cap x (named 'Q))))))
; *. {in N} | N[P] | ().#
; 0. {in N} | N[P] | ().#
; 1. N[P] | M[in N.Q]
; 2. N[M[Q] | P]
#<N[M[Q] | P]>
```

6 Examples With Communication

With communication constructs, some typical exercises become much easier. Here we give two examples that use communication between processes. The first performs a reduction of a tree, and the second is a value cell, reproduced from Cardelli & Gordon (2000).

6.1 Tree Reduction

We define a `tree-reduce` function of four arguments. The first is a function to be called on the leaves of the tree. The second is a function to be called with to combine the results of the reduction on a node's subtrees. The third is a binary tree; if it is a cons cell it has two children, its first and second elements, otherwise it is a leaf. The fourth argument is an ambient name; `tree-reduce` leaves an ambient with the specified name containing an output process that outputs the final value of the tree reduction.

```
(defun tree-reduce (leaf-fn node-fn tree result)
  (if (atom tree)
    (amb result (output (funcall leaf-fn tree)))
    (new (m)
      (amb result
        (input x (input y (output (funcall node-fn x y))))
        (open m) (fun (tree-reduce leaf-fn node-fn (first tree) m))
        (open m) (fun (tree-reduce leaf-fn node-fn (second tree) m))))))
```

For instance, we can negate each leaf in a tree of integers and sum the leaves. We do not show the printed output in this and the next interactive session, as it quickly becomes voluminous.

```
? (run-process
   (tree-reduce '- '+
                '((1 2) (3 (4 5))) 'result))
#<RESULT[{-15}]>
```

We can observe non-determinism by using a non-commutative combination function. For instance, using `list` as the leaf function and `nconc` as the combination function, we obtain a shuffled list of the leaves of the tree. (The list is not randomly ordered. For instance, the two leaf children of a node will always be adjacent in the final list.)

```
? (run-process
   (tree-reduce 'list 'nconc
                '((1 2) (3 (4 5))) 'result))
#<RESULT[{(3 2 4 5 1)}]>

? (run-process
   (tree-reduce 'list 'nconc
                '((1 2) (3 (4 5))) 'result))
#<RESULT[{(1 2 4 3 5)}]>
```

6.2 Value Cells

This cell example depends on the objective movement approximations defined in §4.3 as well as some Lisp macros, as the syntactic sugar that Cardelli & Gordon (2000) give defines binding constructs.

A cell is an ambient that allows entry and exit. Within the ambient, in addition to any processes needed for allowing entry an exit, there is a single output process which will output the value stored by the cell.

```
(defun cell (cell-name value)
  (ambio cell-name (output value)))
```

We want to be able to write `(get c v P)` to bind `v` to the value retrieved from the cell `c` within the body `P`. We do this in two stages. First, we write a function, `get-for`, that takes a cell and a function and invokes the function with the value retrieved from the cell. Second, we write a macro, `get`, that transforms `(get c v P)` into an application of `get-for`.

```
(defun get-for (cell process-function)
  (mv-in cell (input x
                (par (output x)
                     (mv-out cell (funcall process-function x))))))

(defmacro get (cell var &body process)
  `(get-for ,cell #'(lambda (,var) ,@process)))
```

We can check the macroexpansion of `get` to ensure that it expands into a proper call to `get-for`, and indeed, it does. The `get` macro does not save us a great deal of typing over `get-for`, but it allows us to write a way that is more natural and corresponds more directly with the process expression that we aim to encode.

```
? (pprint (macroexpand '(get c v
                          (par (amb v (named 'P))
                               (amb 'n (in v (named 'Q)))))))
(GET-FOR C #'(LAMBDA (V)
               (PAR (AMB V (NAMED 'P))
                    (AMB 'N (IN V (NAMED 'Q))))))
```

Since setting the value of a cell does not involve any binding, we can implement it as a function with no need for a related macro. A value is set by sending a process into the cell's ambient, reading the value, but only to get rid of the output process, creating a new output process, and then leaving the cell's ambient.

```
(defun set (cell value &optional (process (zero)))
  (mv-in cell
         (input x
           (declare (ignore x))
           (par (output value)
                (mv-out cell process)))))
```

The combined get and set operation involves binding again, so we take the same approach that we used for get; we define a function, `get-and-set-in`, that takes the cell to read to and write from, a new value to store in the cell, and function to call with the cell's old value, and then we define a macro `get-and-set` that expands `(get-and-set c v ...)` into an appropriate call to `get-and-set-in`.

```
(defun get-and-set-in (cell value process-function)
  (mv-in cell (input x
                (par (output value)
                     (mv-out cell (funcall process-function x))))))

(defmacro get-and-set (cell var value &body process)
  `(get-and-set-in ,cell ,value #'(lambda (,var) ,@process)))
```

Just to be sure, we can check the macroexpansion for `get-and-set`, just as we did with `get`, and indeed it expands into a proper call to `get-and-set-in`.

```
? (pprint (macroexpand '(get-and-set c v 'n
                          (par (amb v (in 'm))
                               (output 'm)))))
(GET-AND-SET-IN C 'N #'(LAMBDA (V)
                         (PAR (AMB V (IN 'M))
                              (OUTPUT 'M))))
```

7 Advanced Examples

All the examples so far have defined functionality in terms of the Lisp encoding of the ambient calculus. In §2 we made a point of describing exactly how processes are run, and in later sections of observing how evolutions were computed, especially for action processes in §3.3.6. We now consider two extensions to the ambient calculus, viz., true objective movement (in §4.3 we only approximated objective movement) and safe ambients.

7.1 True Objective Moves

Cardelli & Gordon (2000) describe an alternative to the entry and exit capabilities that they adopted for the ambient calculus, an alternative in which action processes move *themselves* into and out of ambients rather than moving their enclosing ambients in and out of ambients. These they represent with *mv in* and *mv out*. We already use `mv-in` and `mv-out` for the approximations, so we use the names `obj-in` and `obj-out` for the true objective capabilities.

We begin with `obj-in`. `obj-in` is a capability, and so we define the class, including `capability` as a superclass. Although not necessary, we also define a constructor for `obj-in` capabilities, `obj-in^`, and a `obj-in` function for creating action processes with `obj-in` capabilities.

```
(defclass obj-in (capability) ()
  (:documentation "Objective entry capability."))

(defun obj-in^ (name)
  (make-instance 'obj-in :name name))

(defun obj-in (name process)
  (cap (obj-in^ name) process))
```

We can construct capabilities and action processes with these functions, but they look a little strange.

```
? (obj-in^ 'x)
#<{capability} X>

? (obj-in 'n (named 'P))
#<{capability} N.P>
```

We can define a `write-process` method for the new capability so that the printed representation will be more descriptive.

```
(defmethod write-process (stream (oi obj-in))
  (format stream "omv in ~:w" (capability-name oi)))

? (obj-in^ 'x)
```

```
#<omv in X>

? (obj-in 'n (named 'P))
#<omv in N.P>
```

In §3.3.6 we described how `evolutions` computes evolutions for action processes: the generic function `action-evolutions` is called with the action process's capability and the action process. Then to get our objective entry working, we need only to define an appropriate `action-evolutions` method.

Our `actions-evolution` method extracts the parent of the action, the action's subprocess, and the name associated with the capability of the action, and initializes an empty list of evolutions. The `dolist` form iterates through the siblings of the action, and if a sibling is an ambient with the appropriate name, and pushes a closure that will put the action's subprocess into the sibling and remove the action from its parent. (Two Lisp-specific points: (i) `(dolist (x y z) ...)` returns the result of evaluating `z`, or in this case, `evolutions`, the list that is built up during the iteration; (ii) the `(let ((sibling sibling)) ...)` is necessary because `dolist` is not required to use separate bindings for each of its iterations, but we need to close over distinct bindings.)

```
(defmethod action-evolutions ((oi obj-in) (action action))
  (let ((parent (process-parent action))
        (aprocess (action-process action))
        (name (capability-name oi))
        (evolutions '()))
    (dolist (sibling (process-siblings action) evolutions)
      (when (and (typep sibling 'ambient)
                 (eql name (ambient-name sibling)))
        (let ((sibling sibling))
          (push #'(lambda ()
                    (setf (process-parent aprocess) sibling)
                    (push aprocess (composition-processes sibling))
                    (setf (composition-processes parent)
                          (delete action
                                  (composition-processes parent))))))))))
```

Indeed, this is all that is necessary to have true objective moves available to us. A simple test shows the expected behavior.

```
(run-process
  (par (obj-in 'm (named 'P))
       (amb 'm (named 'R))))
; *. omv in M.P | M[R]
; 0. omv in M.P | M[R]
; 1. M[P | R]
#<M[P | R]>
```

We do not present the class definition for `obj-out`, though they are included in the examples file, but a suitable `action-evolutions` for objective exit is as follows.

```
(defmethod action-evolutions ((oo obj-out) (action action))
  (let ((parent (process-parent action))
        (aprocess (action-process action)))
    (if (not (amb::ambientp parent (capability-name oo))) '()
      (let ((pparent (process-parent parent)))
        (list #'(lambda ()
                  (setf (process-parent aprocess) pparent)
                  (setf (composition-processes parent)
                        (delete action (composition-processes parent)))
                  (push aprocess (composition-processes pparent))))))))
```

The entrapment example shows the danger associated with true objective moves—ambients can get stuck inside of other ambients with no way to get out!

```
(defun entrap (m)
  (new (k)
    (par (amb k)
         (obj-in m (in k)))))

? (run-process
   (par (entrap 'm)
        (amb 'm (named 'P))))
; *. (#:K[] | omv in M.in #:K) | M[P]
; 0. #:K[] | omv in M.in #:K | M[P]
; 1. #:K[] | M[in #:K | P]
; 2. #:K[M[P]]
#<#:K[M[P]]>
```

The examples file also includes the alternative to *open*, *acid*, which allows a process to dissolve its surrounding ambient.

7.2 Safe Ambients

Levi & Sangiorgi (2003) observe that even without the dangers of truly objective movement, some types of movement in Cardelli & Gordon's (2000) ambient calculus can still cause significant difficulties to running processes. For instance, in the first trace of $open\ n \mid n[in\ m.P] \mid m[Q]$, the ambient n is completely destroyed, in the second n moves into m, out of harm's way.

```
? (run-process
   (par (open 'n)
        (amb 'n (in 'm (named 'P)))
        (amb 'm (named 'Q))))
```

```
; *. open N | N[in M.P] | M[Q]
; 0. open N | N[in M.P] | M[Q]
; 1. in M.P | M[Q]
#<in M.P | M[Q]>

? (run-process
   (par (open 'n)
        (amb 'n (in 'm (named 'P)))
        (amb 'm (named 'Q))))
; *. open N | N[in M.P] | M[Q]
; 0. open N | N[in M.P] | M[Q]
; 1. open N | M[N[P] | Q]
#<open N | M[N[P] | Q]>
```

These difficulties Levi & Sangiorgi (2003) call "grave interferences." We do not consider all the ramifications here, but rather look at their proposed solution, safe ambients. In safe ambients, movement happens only when the ambient into which another ambient would move agrees to the relocation. These agreements from the inside are simply new types of capabilities, and thus we can encode them within the library.

Since we will create six new capabilities, and the definitions will be mostly the same (a class definition, a capability constructor, a `write-process` method, and an action process constructor), we define a macro for defining capabilities.

```
(defmacro defcapability (name)
  "Define a class named NAME that includes capability as a superclass,
a write method for the class, a constructor NAME^ and function NAME
that constructs a process with prefixed with an instance of the new
capability with the specified name."
  (let ((constructor (intern (concatenate 'string (string name) "^")))
        (stream (gensym (string '#:stream-)))
        (cname (gensym (string '#:cname-)))
        (process (gensym (string '#:process-))))
    `(progn
       (defclass ,name (capability)
         ())
       (defmethod write-process (,stream (,name ,name))
         (format ,stream "~(~:w~) ~:w"
                 ',name (capability-name ,name)))
       (defun ,constructor (,cname)
         (make-instance ',name :name ,cname))
       (defun ,name (,cname ,process)
         (cap (,constructor ,cname) ,process)))))
```

We can check the macroexpansion to make sure it is what we expect (or to get a better idea of what exactly it does). `(defcapability $in)` expands as follows.

```
? (pprint (macroexpand '(defcapability $in)))
(PROGN
  (DEFCLASS $IN (CAPABILITY) NIL)
  (DEFMETHOD WRITE-PROCESS (#:STREAM-9479 ($IN $IN))
    (FORMAT #:STREAM-9479 "~(~:w~) ~:w" '$IN (CAPABILITY-NAME $IN)))
  (DEFUN $IN^ (#:CNAME-9480)
    (MAKE-INSTANCE '$IN :NAME #:CNAME-9480))
  (DEFUN $IN (#:CNAME-9480 #:PROCESS-9481)
    (CAP ($IN^ #:CNAME-9480) #:PROCESS-9481)))
```

Using the `defcapability` macro, we define four capabilities. Capabilities beginning with $ request an action and those ending with $ permit it. For instance, our `in$` is, in Levi & Sangiorgi's (2003) terminology, $\overline{\texttt{in}}$.

```
(defcapability $in)
(defcapability in$)
(defcapability $out)
(defcapability out$)
(defcapability $open)
(defcapability open$)

? (par (amb 'n ($in 'm (named 'P)))
       (amb 'm (in$ 'm (named 'Q))))
#<N[$in M.P] | M[in$ M.Q]>
```

Now we need only define appropriate `action-evolutions` methods for the new capabilities to get safe ambients running in our system. As with the output communication constructs developed in §5, we have matching pairs of processes, and we will only define `action-evolutions` methods for one side of the pairs. Particularly, we define methods for `$in`, `$out`, and `$open`.

7.2.1 Safe Entry

We tend to safe entry first. It will be convenient to check whether an process is an action process allowing entry for a given name; this is handled by `entry-allower-p`.

```
(defun entry-allower-p (name process)
  (and (typep process 'action)
       (typep (action-capability process) 'in\$)
       (eql (capability-name (action-capability process)) name)))
```

Now we can define the `actions-evolutions` method for `$in` actions. This must check whether the action is enclosed in an ambient, and if it is whether the enclosing ambient has any siblings that are ambients with the relevant name, and whether those ambients have any action processes within them that allow entry. As when we developed true objective moves in

§7.1, we use `dolist` to return the incrementally built up evolutions list, and when we push the closures into the list, we rebind `p` to ensure that we close over a unique binding for the desired process. The function `amb::ambientp` checks whether an object is an ambient and, if a second argument is provided, whether the ambient has a given name. `amb::move-ambient` removes its first argument, an ambient, from its parent and adds it to its second argument, another ambient. Since `entry-allower-p` is unlikely to be used anywhere else, we define it as a local function with `flet`.

```
(defmethod action-evolutions (($in $in) (action action))
  (if (not (amb::ambientp (process-parent action))) '()
    (let ((mover (process-parent action))
          (target-name (capability-name (action-capability action)))
          (evolutions '()))
      (flet ((entry-allower-p (x)
               (and (typep x 'action)
                    (typep (action-capability x) 'in\$)
                    (eql target-name
                         (capability-name (action-capability x))))))
        (dolist (sibling (process-siblings mover) evolutions)
          (when (amb::ambientp sibling target-name)
            (dolist (p (composition-processes sibling))
              (when (entry-allower-p p)
                (let ((p p)
                      (sibling sibling))
                  (push #'(lambda ()
                            (amb::deprefix action)
                            (amb::deprefix p)
                            (amb::move-ambient mover sibling))
                        evolutions)))))))))))
```

(It might be a useful exercise to revisit §7.1 to see where `amb::ambientp`, `amb::deprefix`, and `amb::move-ambient` could have been used in defining the `actions-evolutions` methods for the objective movements. We refrained from using these functions there in order to demonstrate a complete example.)

We now have safe entry and can run the process we constructed earlier. We can also run a variant of the process that cannot progress because the target has no allowing process within it.

```
? (run-process
   (par (amb 'n ($in 'm (named 'P)))
        (amb 'm (in$ 'm (named 'Q)))))
; *. N[$in M.P] | M[in$ M.Q]
; 0. N[$in M.P] | M[in$ M.Q]
; 1. M[N[P] | Q]
#<M[N[P] | Q]>
```

```
? (run-process
   (par (amb 'n ($in 'm (named 'P)))
        (amb 'm (named 'Q))))
; *. N[$in M.P] | M[Q]
; 0. N[$in M.P] | M[Q]
#<N[$in M.P] | M[Q]>
```

7.2.2 Safe Exit

Safe exit only requires another `action-evolutions` method, and it is not all that different from the method for safe entry. Recall that safe exit progresses allows, for instance, $m[n[\$\text{out } m.P] \mid \text{out\$ } m.Q]$ to evolve as $n[P] \mid m[Q]$. Then `actions-evolutions` for an `$out` capability should check that the closing composition is an ambient inside an ambient, and that there is a sibling allowing ambients to leave.

```
(defmethod action-evolutions (($out $out) (action action))
  ;; m[n[$out m.P] | out$ m.Q] => n[P] | m[Q]
  (let ((n (process-parent action))
        (m (capability-name $out))
        (evolutions '()))
    (flet ((out-allower-p (x)
             (and (typep x 'action)
                  (typep (action-capability x) 'out$)
                  (eql m (capability-name (action-capability x))))))
      (when (and (amb::ambientp n)
                 (amb::ambientp (process-parent n) m))
        (dolist (sibling (process-siblings n) evolutions)
          (when (out-allower-p sibling)
            (let ((sibling sibling))
              (push #'(lambda ()
                        (amb::deprefix action)
                        (amb::deprefix sibling)
                        (amb::move-ambient
                         n (process-parent (process-parent n))))
                    evolutions)))))))))
```

As with safe entry, we can see how safe exit may proceed, and also when it may not.

```
? (run-process
   (par (amb 'm
          (amb 'n ($out 'm (named 'P)))
          (out$ 'm (named 'Q)))))
; *. M[N[$out M.P] | out$ M.Q]
; 0. M[N[$out M.P] | out$ M.Q]
; 1. N[P] | M[Q]
```

```
#<N[P] | M[Q]>

? (run-process
   (par (amb 'm
          (amb 'n ($out 'm (named 'P)))
          (named 'Q))))
; *. M[N[$out M.P] | Q]
; 0. M[N[$out M.P] | Q]
#<M[N[$out M.P] | Q]>
```

7.2.3 Safe Open

Safe ambient opening is the last safe ambient capability to implement. Safe opening ensures that only ambients willing to be opened may be opened. A process $open $n.P \mid n[\text{open\$}\ n.Q]$ evolves as $P \mid Q$.

```
(defmethod action-evolutions (($open $open) (action action))
  (let ((n (capability-name $open))
        (evolutions '()))
    (flet ((open-allower-p (x)
             (and (typep x 'action)
                  (typep (action-capability x) 'open$)
                  (eql n (capability-name (action-capability x))))))
      (dolist (sibling (process-siblings action) evolutions)
        (when (amb::ambientp sibling n)
          (dolist (p (composition-processes sibling))
            (when (open-allower-p p)
              (let ((p p)
                    (sibling sibling))
                (push #'(lambda ()
                          (amb::deprefix action)
                          (amb::deprefix p)
                          (change-class sibling 'composition))
                      evolutions)))))))))
```

The `actions-evolutions` method for safe opening is very similar to the other methods we have defined. One notable point, though, is the use of `change-class`, which changes the class of a CLOS instance. `change-class` turns the ambient into a composition, discarding its name, but preserving its composed processes. The `cleanup` method that is invoked after every evolution will automatically flatten any nested compositions, including the one that the ambient just changed into. We can see examples of safe opening evolving or not, as before.

```
? (run-process
   (par ($open 'n (named 'P))
        (amb 'n (open$ 'n (named 'Q)))))
```

```
; *. $open N.P | N[open$ N.Q]
; 0. $open N.P | N[open$ N.Q]
; 1. P | Q
#<P | Q>

? (run-process
   (par ($open 'n (named 'P))
        (amb 'n (named 'Q))))
; *. $open N.P | N[Q]
; 0. $open N.P | N[Q]
#<$open N.P | N[Q]>
```

In this section we have seen that it is relatively easy to extend the system with new types of capabilities that evolve in ways distinct from the original ambient calculus.

8 Future Work

Though the system as we have it now is sufficiently developed to encode the ambient calculus as well as some of its variants and extensions, there are still number a number of improvements to implement and enhancements to explored. We consider three of these here.

8.1 Removing Inert Processes

The `cleanup` method used by `run-process` simplifies many processes in important ways, flattening nested compositions and removing instances of the inactive process, **0**. There are other kinds of processes, however, that can justifiably be removed that currently remain in processes. For instance, in the following example, there is an ambient named by a fresh symbol which is not accessible (via usual means in the ambient calculus) to any other process.

```
? (run-process
   (par (new (a) (amb a))
        (named 'p)))
; *. #:A[] | P
; 0. #:A[] | P
#<#:A[] | P>
```

Ideally the final process in the trace could evolve one more time, producing just P. In the current implementation, recognizing that the ambient could be removed would require tracking the usage of fresh names, and this is an unappealing task. With more and more Lisps supporting weak pointers (i.e., pointers to objects that do not prevent a garbage collector from collecting the object) it may be possible to encode ambients in such a way that ambients like the one above could be removed, not by a `cleanup` method, but by the system garbage collector. (In fact, this task can only be correctly handled by the system garbage collector because names may be referenced by arbitrary non-process objects.)

8.2 Concurrency

The current library provides an easy way to observe non-deterministic interleaving of process execution, but lacks genuine concurrency. Many Lisps have support for multi-threading, and it is probably not too difficult to associate OS (or lightweight) threads with processes in the ambient calculus. Even so, this might not have the desired effect. For instance, in LispWorks 5.1, "Each Lisp `mp:process` has a separate native thread. You can have many runnable mp:process objects/native threads, but Lisp code can only run in one thread at a time and a lock is used to enforce this. This can limit performance on multi-CPU machines."[1]

Additionally, not every ambient should correspond to a thread of execution or a theatre for process execution. For instance, the semaphores described in §4.1 use ambients as semaphores, and these ambients never contain any processes, nor do they ever move, and so on. Their only purpose is to exist as observable objects. Of course, that particular encoding of semaphores was designed to illustrate the power of the ambient calculus, and there may be no reason not to use a different kind of locking mechanism in real applications.

8.3 Distribution

The calculus of mobile ambients was developed to formalize and reason about mobile processes, i.e., relocatable processes that can be spread over a network. The current system has no support for distribution. Concurrency is necessary before distribution can be approached, but distribution will require significant efforts beyond concurrency. Many of the techniques used in implemented the system, e.g., lexical closures over an environment, are not suitable in a distributed setting. That issue notwithstanding, many distributed systems have already addressed implementation issues, and so their techniques are almost certainly directly applicable. Thus, while distribution requires lots of work, much of that work has already been done elsewhere, and needs only to be ported to the current system.

9 For More Information

Documentation for the library is available online at the project homepage, and the author can be contacted at `tayloj@cs.rpi.edu`.

10 Related Work

There is now an abundance of mobile ambient literature and we make no attempt to enumerate it here. There have been other implementation efforts for mobile ambients and some of its variants, and we do provide some pointers to these here.

Pous (2002) produced an graphical simulator for mobile ambients. The simulator is written in Java, and can be run as an applet within a web browser, or as an application,

[1] `http://www.lispworks.com/documentation/lw51/LWUG/html/lwuser-238.htm`

by downloading `ambicobj.jar`. Several ambient variants are implemented, viz.: mobile ambients, safe ambients, robust ambients, and controlled ambients. Users may enter ambients expressions interactively and watch the resulting system evolve. The input syntax follows the syntax of ambients in the formal theory, extended with a few primitives for practical purposes. For instance, an example from Zimmer (2002) shows `a[in b.sleep 100.out b]` as the input text for the ambient expression $a[in\ b.sleep\ 100.out\ b]$ where *sleep* is a new capability for causing a delay. Replication does not seem to be supported directly; as in the present library, explicit recursion is available. Another example from the same source shows `x[rec X.x[out x.X]]` as an example of recursion; the capability `rec X` allows the prefixed action to refer to itself as `X`. The source does not seem to be available from the project webpage, and it is unclear whether arbitrary Java code can be introduced to ambient behaviors.

Fournet, Lévy & Schmitt (2000) describe an implementation of mobile ambients in JoCaml. Their implementation is based on translation, which they prove correct, from mobile ambients to the distributed join calculus. Join calculus expressions in hand, they were able to use a (now unmaintained) implementation of JoCaml (Conchon & Fessant 1999) with support for distribution to realize a distributed ambient system.

References

Cardelli, L. & Gordon, A. D. (2000), 'Mobile ambients', *Theoretical Computer Science* **240**(1), 177–213.
URL: http://dx.doi.org/10.1016/S0304-3975(99)00231-5

Conchon, S. & Fessant, F. L. (1999), Jocaml: Mobile Agents for Objective-Caml, *in* 'First International Symposium on Agent Systems and Applications 1999, and Third International Symposium on Mobile Agents', pp. 22–29.
URL: http://dx.doi.org/10.1109/ASAMA.1999.805390

Fournet, C., Lévy, J.-J. & Schmitt, A. (2000), An asynchronous, distributed implementation of mobile ambients, *in* J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses & T. Ito, eds, 'Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics', Vol. 1872 of *Lecture Notes in Computer Science*, Springer, pp. 348–364.
URL: http://dx.doi.org/10.1007/3-540-44929-9

Levi, F. & Sangiorgi, D. (2003), 'Mobile safe ambients', *ACM Transactions on Programming Languages and Systems* **25**(1), 1–69.
URL: http://doi.acm.org/10.1145/596980.596981

Pous, D. (2002), Les mobile ambients en Icobj, Intership report, INRIA Sophia Antipolis.
URL: http://www-sop.inria.fr/mimosa/ambicobjs/rapport.ps

Zimmer, P. (2002), 'The AmbIcobjs demo: Ambient programming in Icobjs', http://www-sop.inria.fr/mimosa/ambicobjs/.
URL: http://www-sop.inria.fr/mimosa/ambicobjs/