Goblins: a transactional, distributed actor model environment

Version 7.6

Christopher Lemmer Webber

March 5, 2020

CAUTION: *Goblins is currently alpha. Semantics may change and you may be forced to update your code. Be mindful of that before using this for anything that goes into production!*

Contents

1 What is Goblins?

Goblins is a quasi-functional distributed object system, mostly following the actor model. Its design allows for object-capability security, allowing for safe distributed programming environments.[1] Its design is inspired by the E programming language and by the observation that lambda is already the ultimate security mechanism (ie, normal argument-passing in programs, if taken seriously/purely, is already all the security system we need).

1.1 What works so far?

- **Quasi-functional object system:** Users hold on to references of objects/actors, but objects/actors are just procedures. Rather than directly mutating, objects/actors can specify that they should "become" a new version of themselves when they handle the next invocation.
- **Transactional updates:** Changes happen within a transaction. If an unhandled exception occurs, we can "roll back" history as if the message never happened, avoiding confused state changes throughout the system.
- **Time travel:** We can snapshot old revisions of the system and interact with them.
- **Asynchronous programming with sophisticated promise chaining:** Asynchronous message passing with promises is supported. Promises work such that there's no need to wait for promises to resolve before interacting... you can communicate with the future that doesn't even yet exist! Sent a message to that remote car factory asking for a new car? Why wait for the car to be delivered... you can send it a drive message at the same time, and both messages can be delivered across the network in a single hop.
- **Communicating event loops:** Objects across multiple event loops (aka "vats") can communicate, whether in the same OS process or (soon) across the network.
- **Synchronous and asynchronous behavior, integrated but distinguished*:** Both synchronous and asynchronous programming is supported, but only objects in the same "vat" (event loop) can perform synchronous immediate calls. All objects can perform asynchronous calls against each other.
- **A library, not a language:** Goblins is itself a library that can be utilized with nearly any Racket program, including many Racket #langs. (However, some languages may be provided or encouraged for additional security / convenience later).

[1]By now you may have noticed that we've used the term "object" twice... we may not be meaning the term the way you think! In the early to mid 2000s, "object oriented programming" was all the rage such that being reprimanded by an OOP zealots for not fitting into their worldviews was common. These days the pendelum has swung in the opposite direction such that one is more likely to be reprimanded by *anti*-object-oriented zealots. In fact, "Object Oriented" has many different meanings. If you have PTSD from complex Java or C++ class structures, please read on... our use of the term doesn't involve any fundamental form of "classes" or "inheritance". Furthermore, "object" in "object capability security" refers to security based on holding onto references to an external entity. What you hold onto is what you can do. Even most purely functional languages can fit into that definition, and so too can most functional languages follow object capability security.

- **Object capability security:** Goblins itself is built for object capability (ocap) security, which is to say that you can only operate on the references you have access to. Goblins embraces and builds upon this foundation. (However, full ocap security will require a safer module system than the one Racket provides; coming eventually.)

1.2 What's on its way?

- **Fully distributed, networked, secure p2p communication:** In the future, it will be possible to communicate with other objects over a network connection. Distributed object interactions is safe because of ocap security guarantees.
- **A separate module system:** While not part of Goblins itself, a future project named "Spritely Dungeon" will help close the loop on ocap security for Racket.

2 A tutorial

2.1 Getting going

First you will need to download and install Racket, if you haven't already. You will want an editor that you can comfortably program Racket in. If you are already a user of GNU Emacs, racket-mode is most excellent. If you aren't already an Emacs user, Racket ships with an excellent built-in IDE called DrRacket. Use that.

Next you will need to install Goblins. At the command line you can type:

```
raco pkg install goblins
```

Alternately, from DrRacket you can go to `File => Install Package`.

If you are totally new to Racket, see the Getting Started documentation page. Otherwise, let's go!

2.2 Vats, actors, spawning, and immediate calls

Open a Racket REPL and import Goblins:

```
#lang racket
(require goblins)
```

First we're going to need something to store our objects in. We'll boot up an event loop, called a vat (TODO: explain why it's called that), which can manage our objects.

```
> (define a-vat
    (make-vat))
```

Our vat is currently lonely... nobody lives in it! Let's make a friend. First we'll need a friend constructor:

```
;; The ^ is conventionally called a "hard hat" in Goblins; it means
;; this is an object constructor.
;; Every constructor takes a bcom argument, the rest of them are
;; passed in from the spawn invocation.
(define (^friend bcom my-name)
  ;; This is the initial handler procedure.
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))
```

The outer procedure is the constructor; all it really does is return another procedure which is the handler.

Let's make a friend and call her Alice.

```
(define alice
  (a-vat 'spawn ^friend "Alice"))
```

Here the arguments to the spawn method are `^friend`, which is the constructor procedure we are using, and the argument "Alice", which becomes bound to `my-name`. (The `bcom` argument is implicitly provided by Goblins; we'll ignore it for right now.)

If we look at Alice in the REPL, we'll see that what we're really holding onto is a "live reference" to Alice.

```
> alice
#<live-refr ^friend>
```

Now we'd like to talk to Alice. For now let's use the "call" method on our vat:

```
> (a-vat 'call alice "Chris")
"Hello Chris, my name is Alice!"
```

If we look at our `^friend` procedure again, this should make a lot of sense; the inner lambda is being evaluated with "Chris" being passed in for `your-name`. The returned value is just the result.

Normally in a Goblins program, we can just spawn and talk to a Goblins object directly using the `spawn` and `$` operators directly. However we're kind of "bootstrapping the world" here, so we need the vat's help to do that. However, we can see what it would look like to use them if we were in a "Goblins context" by using the vat's `'run` method which allows us to pass in an arbitrary thunk (aka "procedure with no arguments"):

```
> (a-vat 'run
         (lambda ()
           (define alyssa
             (spawn ^friend "Alyssa"))
           ($ alyssa "Ben")))
"Hello Ben, my name is Alyssa!"
```

Anyway, maybe we'd like to be greeted the same way we have been by our friend sometimes, but other times we'd like to just find out what our friend's name is. It would be nice to have different "methods" we could call, and in fact, Goblins comes with a convenient `methods` macro:

```
(require goblins/actor-lib/methods)

(define (^methods-friend bcom my-name)
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)]
   ;; return what our name is
   [(name)
    my-name]))
```

Now let's spawn an alice2 that uses `^methods-friend`:

```
(define alice2
  (a-vat 'spawn ^methods-friend "Alice"))
```

Now we can call each method separately:

```
> (a-vat 'call alice2 'name)
"Alice"
> (a-vat 'call alice2 'greet "Chris")
"Hello Chris, my name is Alice!"
```

(As a side note, if you're thinking that it would look nicer if this looked like:)

```
(a-vat.call alice2.name)
(a-vat.call alice2.greet "Chris")
```

(... you're right and a `#lang` will provided in the future for such aesthetic improvement which expands to the former syntax.)

What kind of magic is this methods thing? Well actually it's barely any magic at all. Methods just returns a procedure that dispatches on the first argument, a symbol, to one of several procedures. We can even use it outside of an object/actor constructor:

```
> (define what-am-i
    (methods
     [(i-am)
      (list 'i 'am 'just 'a 'procedure)]
     [(that what)
      (list 'that 'calls 'other what)]))
> (what-am-i 'i-am)
'(i am just a procedure)
> (what-am-i 'that 'procedures)
'(that calls other procedures)
```

We could have just as well written methods-friend like so:

```
(define (^match-friend bcom my-name)
  (match-lambda*
   ;; Greet the user using their name
   [(list 'greet your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)]
   ;; return what our name is
   [(list 'name)
    my-name]))
```

But it's a lot nicer to use `methods`. But the key thing to realize is that `methods` just itself returns another procedure.

Maybe we'd like to keep track of how many times our friend has been called. It might be helpful to have some kind of helper object which can do that. What if we made a counter?

```
(define (^counter bcom [count 0])
  (methods
   ;; return the current count
   [(count)
    count]
   ;; Add one to the current counter
   [(add1)
    (bcom (^counter bcom (add1 count)))]))
```

Now let's spawn and poke at an instance of that counter a bit:

```
> (define a-counter
    (a-vat 'spawn ^counter))
> (a-vat 'call a-counter 'count)
0
> (a-vat 'call a-counter 'add1)
> (a-vat 'call a-counter 'count)
1
> (a-vat 'call a-counter 'add1)
> (a-vat 'call a-counter 'add1)
> (a-vat 'call a-counter 'count)
3
```

Now note that our counter actually appears to change... how does this happen? Let's look at the body of that add1 method in detail:

```
(bcom (^counter bcom (add1 count)))
```

bcom (pronounced "be-come" or "be-comm") is the capability to become a new version of ourselves; more explicitly, to give a procedure which will be called the *next time* this object is invoked. next returns some methods wrapped up in a closure which knows what the count is; we're incrementing that by one from whatever we currently have. (Depending on how experienced you are with functional programming is how confusing this is likely to be.)

There are multiple equivalent ways we could build the "next" procedure we are becoming for ourselves, and one is to build an actual next builder and instantiate it once. This technique is exactly equivalent to the above, and we will use this kind of structure later, so it's worth seeing and realizing it's more or less the same (except that we didn't expose the choice of an initial count value):

```
(define (^counter bcom)
  (define (next count)
    (methods
     ;; return the current count
     [(count)
      count]
     ;; Add one to the current counter
     [(add1)
      ;; Become the next version of ourselves,
      ;; with count incremented
      (bcom (next (add1 count)))]))
  ;; We'll start at 0.
  (next 0))
```

Now that we have this counter, we can rewrite our friend to spawn it and use it:

```
(define (^counter-friend bcom my-name)
  (define greet-counter
    (spawn ^counter))
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    ;; Increment count by one, since we were just called.
    ;; The counter starts at 0 so this will be correct.
    ($ greet-counter 'add1)
    (define greet-count
      ($ greet-counter 'count))
    (format "Hello ~a, my name is ~a and I've greeted ~a times!"
            your-name my-name greet-count)]
   ;; return what our name is
   [(name)
    my-name]
   ;; check how many times we've greeted, without
   ;; incrementing it
```

```
    [(greet-count)
     ($ greet-counter 'count)]))
```

You'll observe that there was no need to go through the vat here, our object was able to use spawn and $ (which is pronounced "call", "immediate call", or "money call") directly. That's because our actor is operating within a goblins context, so there's no reason to do so by bootstrapping through the vat (indeed, trying to do so would cause an exception to be raised).

Now let's give it a try:

```
> (define alice3
    (a-vat 'spawn ^counter-friend "Alice"))
> (a-vat 'call alice3 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 1 times!"
> (a-vat 'call alice3 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 2 times!"
> (a-vat 'call alice3 'greet-count)
2
> (a-vat 'call alice3 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 3 times!"
> (a-vat 'call alice3 'greet-count)
3
```

Perhaps we'd like to have our friend remember the last person she was called by. It would be nice if there were something along the lines of a mutable variable we could change. In fact there is, and it's called a cell. When called with no arguments, a cell returns its current value. When called with one argument, a cell replaces its current value with the one we have provided:

```
> (require goblins/actor-lib/cell)
> (define treasure-chest
    (a-vat 'spawn ^cell 'gold))
> (a-vat 'call treasure-chest)
'gold
> (a-vat 'call treasure-chest 'flaming-sword)
> (a-vat 'call treasure-chest)
'flaming-sword
```

A fun exercise is to try to write your own cell. Here is one way:

```
;; Constructor for a cell.  Takes an optional initial value, defaults
;; to false.
(define (^our-cell bcom [val #f])
  (case-lambda
    ;; Called with no arguments; return the current value
    [() val]
    ;; Called with one argument, we become a version of ourselves
    ;; with this new value
    [(new-val)
     (bcom (^our-cell bcom new-val))]))
```

Of course, you could also have a cell that instead has 'get and 'set methods. This is left as an exercise for the reader.

Now that we have cells, we can use them:

```
(define (^memory-friend bcom my-name)
  (define greet-counter
    (spawn ^counter))
  (define recent-friend
    (spawn ^cell #f))
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    ;; Increment count by one, since we were just called.
    ;; The counter starts at 0 so this will be correct.
    ($ greet-counter 'add1)
    (define greet-count
      ($ greet-counter 'count))
    ;; Who our friend was last time
    (define last-friend-name
      ($ recent-friend))
    ;; But now let's set the recent friend to be this name
    ($ recent-friend your-name)
    (if last-friend-name
        (format "Hello ~a, my name is ~a and I've greeted ~a times (last by ~a)!"
                your-name my-name greet-count last-friend-name)
        (format "Hello ~a, my name is ~a and I've greeted ~a times!"
                your-name my-name greet-count))]
   ;; return what our name is
   [(name)
    my-name]
   ;; check how many times we've greeted, without
   ;; incrementing it
   [(greet-count)
    ($ greet-counter 'count)]))
```

Let's try interacting with this friend:

```
> (define alice4
    (a-vat 'spawn ^memory-friend "Alice"))
> (a-vat 'call alice4 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 1 times!"
> (a-vat 'call alice4 'greet "Carl")
"Hello Carl, my name is Alice and I've greeted 2 times (last by
Chris)!"
> (a-vat 'call alice4 'greet "Carol")
"Hello Carol, my name is Alice and I've greeted 3 times (last by
Carl)!"
```

Whew... if we look at that code for the `greet` code, it sure looks fairly imperative, though. We pulled out the value from `recent-friend` before we changed it.

If we had accidentlaly put the definition for `last-friend-name` after setting `recent-friend` to `your-name`, we might have resulted in a classic imperative error and `last-friend-name` would be set to the new one instead of the old one!

Well, it turns out that `bcom` can take a second argument which provides a value it would like to return in addition to specifying the new version of itself it would like to become. This means that we could rewrite `^memory-friend` like so with no behavioral differences:

```
(define (^memory-friend2 bcom my-name)
  (define (next greet-count last-friend-name)
    (methods
     ;; Greet the user using their name
     [(greet your-name)
      (define greeting
        (if last-friend-name
            (format "Hello ~a, my name is ~a and I've greeted ~a times (last by ~a)!"
                    your-name my-name greet-count last-friend-
name)
            (format "Hello ~a, my name is ~a and I've greeted ~a times!"
                    your-name my-name greet-count)))
      (bcom (next (add1 greet-count)
                  your-name)
            greeting)]
     ;; return what our name is
     [(name)
      my-name]
     ;; check how many times we've greeted, without
     ;; incrementing it
     [(greet-count)
      greet-count]))
  (next 1 #f))

> (define alice5
    (a-vat 'spawn ^memory-friend2 "Alice"))
> (a-vat 'call alice5 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 1 times!"
> (a-vat 'call alice5 'greet "Carl")
"Hello Carl, my name is Alice and I've greeted 2 times (last by
Chris)!"
> (a-vat 'call alice5 'greet "Carol")
"Hello Carol, my name is Alice and I've greeted 3 times (last by
Carl)!"
```

This certainly looks more functional, and we have some freedom of how we'd like to implement it. It also leads to the observation that the behavior of objects in respect to updating themselves appears to be very functional (returning a new version of ourselves and maybe a value), whereas calling other objects appears to be very imperative. So what is the point of cells and counters? After all, if we're using `#lang racket` we have access to the `set!` procedure, and could have easily rewritten the `^counter` and `^cell` versions like so:

```
(define (^imperative-friend bcom my-name)
  (define greet-count
    0)
  (define recent-friend
    #f)
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    ;; Increment count by one, since we were just called.
    ;; The counter starts at 0 so this will be correct.
    (set! greet-count (add1 greet-count))
    ;; Who our friend was last time
    (define last-friend-name
      recent-friend)
    ;; But now let's set the recent friend to be this name
    (set! recent-friend your-name)
    (if last-friend-name
        (format "Hello ~a, my name is ~a and I've greeted ~a times (last by ~a)!"
                your-name my-name greet-count last-friend-name)
        (format "Hello ~a, my name is ~a and I've greeted ~a times!"
                your-name my-name greet-count))]
   ;; return what our name is
   [(name)
    my-name]
   ;; check how many times we've greeted, without
   ;; incrementing it
   [(greet-count)
    greet-count]
   [(recent-friend)
```

```
      recent-friend]))
```

Usage is exactly the ssame:

```
> (define alice6
    (a-vat 'spawn ^imperative-friend "Alice"))
> (a-vat 'call alice6 'greet "Chris")
"Hello Chris, my name is Alice and I've greeted 1 times!"
> (a-vat 'call alice6 'greet "Carl")
"Hello Carl, my name is Alice and I've greeted 2 times (last by
Chris)!"
> (a-vat 'call alice6 'greet "Carol")
"Hello Carol, my name is Alice and I've greeted 3 times (last by
Carl)!"
```

This code looks mostly the same too, and indeed maybe even a little simpler with `set!` (no mucking around with that `$` malarky).

Let's introduce a couple of bugs into both the cell-using and the `set!` using imperative versions of these friends:

```
(define (^buggy-memory-friend bcom my-name)
  (define greet-counter
    (spawn ^counter))
  (define recent-friend
    (spawn ^cell #f))
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    ;; Increment count by one, since we were just called.
    ;; The counter starts at 0 so this will be correct.
    ($ greet-counter 'add1)
    (define greet-count
      ($ greet-counter 'count))
    ;; Who our friend was last time
    (define last-friend-name
      ($ recent-friend))
    ;; But now let's set the recent friend to be this name
    ($ recent-friend your-name)
    (error "AHH! Throwing an error after I changed things!")
    (if last-friend-name
        (format "Hello ~a, my name is ~a and I've greeted ~a times (last by ~a)!"
                your-name my-name greet-count last-friend-name)
        (format "Hello ~a, my name is ~a and I've greeted ~a times!"
                your-name my-name greet-count))]
    ;; return what our name is
    [(name)
     my-name]
    ;; check how many times we've greeted, without
    ;; incrementing it
    [(greet-count)
     ($ greet-counter 'count)]))

(define (^buggy-imperative-friend bcom my-name)
  (define greet-count
    0)
  (define recent-friend
    #f)
  (methods
   ;; Greet the user using their name
   [(greet your-name)
    ;; Increment count by one, since we were just called.
    ;; The counter starts at 0 so this will be correct.
    (set! greet-count (add1 greet-count))
    ;; Who our friend was last time
    (define last-friend-name
      recent-friend)
    ;; But now let's set the recent friend to be this name
    (set! recent-friend your-name)
    (error "AHH! Throwing an error after I changed things!")
    (if last-friend-name
        (format "Hello ~a, my name is ~a and I've greeted ~a times (last by ~a)!"
                your-name my-name greet-count last-friend-name)
        (format "Hello ~a, my name is ~a and I've greeted ~a times!"
                your-name my-name greet-count))]
   ;; return what our name is
   [(name)
    my-name]
   ;; check how many times we've greeted, without
   ;; incrementing it
   [(greet-count)
    greet-count]))

(define buggy-gobliny-alice
  (a-vat 'spawn ^buggy-memory-friend "Alice"))
(define buggy-imperative-alice
  (a-vat 'spawn ^buggy-imperative-friend "Alice"))
```

(Observe the `error` put after both of them changed `greet-count` and `recent-friend`.)

Okay, first let's check the initial `greet-count`:

```
> (a-vat 'call buggy-gobliny-alice 'greet-count)
0
> (a-vat 'call buggy-imperative-alice 'greet-count)
0
```

So far, so good. Now let's greet both of them:

```
> (a-vat 'call buggy-gobliny-alice 'greet "Chris")
AHH! Throwing an error after I changed things!
> (a-vat 'call buggy-imperative-alice 'greet "Chris")
AHH! Throwing an error after I changed things!
```

Okay, so both of them threw the error. But what do you think the result of =greet-count= will be now?

```
> (a-vat 'call buggy-gobliny-alice 'greet-count)
0
> (a-vat 'call buggy-imperative-alice 'greet-count)
1
```

Now this is definitely different! In the goblin'y example, by using goblin objects/actors, unhandled errors means that breakage is as if nothing ever occurred. We can log the error, but we won't mess up the rest of the system. With the imperative code which uses =set!=, the state of our system could become unintentionally corrupted and inconsistent.

This is what we mean by Goblins being transactional: something that goes wrong need not be "committed" to the current state. This is important for systems like financial infrastructure. It turns out it also opens us up, in general, to becoming {time wizards}. But more on that later.

2.3 Message passing, promises, and multiple vats

2.3.1 The basics

Remember simpler times, when friends mostly just greeted us hello?

```
(define (^friend bcom my-name)
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))

(define alice
  (a-vat 'spawn ^friend "Alice"))
```

We could of course make another friend that talks to Alice.

```
(define (^calls-friend bcom our-name)
  (lambda (friend)
    (define what-my-friend-said
      ($ friend our-name))
    (displayln (format "<~a>: I called my friend, and they said:"
                       our-name))
    (displayln (format "   \"~a\"" what-my-friend-said))))

(define archie
  (a-vat 'spawn ^calls-friend "Archie"))
```

Now Archie can talk to Alice:

```
> (a-vat 'call archie alice)
<Archie>: I called my friend, and they said:
   "Hello Archie, my name is Alice!"
```

Both Alice and Archie live in `a-vat`. But `a-vat` isn't the only vat in town. One other such vat is `b-vat`, where Bob lives:

```
(define b-vat
  (make-vat))

(define bob
  (b-vat 'spawn ^calls-friend "Bob"))
```

Obviously, since Bob is in `b-vat`, we bootstrap a message call to Bob from `b-vat`. But what do you think happens when Bob tries to call Alice?

```
> (b-vat 'call bob alice)
not-callable: Not in the same vat: #<live-refr ^friend>
```

Oh no! It looks like Bob can't call Alice since they live in different places! From Archie's perspective, Alice was "near", aka "in the same vat". However from Bob's perspective Alice was "far", aka "in some other vat that isn't the one I'm in". This is a problem because using the `$` operator performs a *synchronous* call, but it's only safe to do synchronous calls for objects that are near each other (in the same vat).

Fortunately there's something we can do: we can send a message from Bob to Alice. But we've never seen message sending in Goblins before, so what is that?

To prototype this, let's use the `'run` method on `b-vat`. Remember, we saw the `'run` method used before, where it looked like:

```
> (a-vat 'run
         (lambda ()
           (define alyssa
             (spawn ^friend "Alyssa"))
           ($ alyssa "Ben")))
"Hello Ben, my name is Alyssa!"
```

So 'run is just a way to run some arbitrary code in an actor context. That sounds good enough for playing around with sending messages. We can send messages with `<-` so let's try that:

```
> (b-vat 'run
         (lambda ()
           (<- alice "Brenda")))
#<live-refr promised>
```

Ah ok... so what `<-` returns is something called a "Promise" which might eventually be resolved to something interesting. We want some way to be able to pull out that interesting thing. That's what `on` is for: it resolves promises and pulls out their resolved value:

```
> (b-vat 'run
         (lambda ()
           (on (<- alice "Brenda")
               (lambda (alice-says)
                 (displayln (format "Got from Alice: ~a" alice-
says))))))
Got from Alice: Hello Brenda, my name is Alice!
```

`<-` works just fine with far references, but it also works just fine with near references too! So we can run the same code in a-vat (where Alice is "near") and it works there too:

```
> (a-vat 'run
         (lambda ()
           (on (<- alice "Arthur")
               (lambda (alice-says)
                 (displayln (format "Got from Alice: ~a" alice-
says))))))
Got from Alice: Hello Arthur, my name is Alice!
```

So using `on` and `<-` seems to fit our needs. But what would have happened if Alice had thrown an error? Indeed, if we remember earlier we made `buggy-gobliny-alice` so we can test for that. It turns out that on can take a `#:catch` argument:

```
> (b-vat 'run
          (lambda ()
            (on (<- buggy-gobliny-alice 'greet "Brenda")
                (lambda (alice-says)
                  (displayln (format "Got from Alice: ~a" alice-
says)))
                #:catch
                (lambda (err)
                  (displayln "Tried to talk to Alice, got an error
:("))))))
Tried to talk to Alice, got an error :(
;; === While attempting to send message: ===
AHH! Throwing an error after I changed things!
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

Now this is a little bit confusing to read because we saw two separate messages here... it's important to realize that due to the way our vat is configured, the exception backtrace being printed out is coming from `a-vat`, not from our code being evaluated in `b-vat`. We could configure the `a-vat` loop to do something different when it hits errors, but currently it prints exceptions so we can debug them. Anyway, so that's helpful information, but actually the place we caught the error in *our* code above was in the `lambda` right after `#:catch`. As we can see, it did catch the error and we used that as an opportunity to print out a complaint.

So `<-` makes a promise for us. We don't always need a promise; sometimes we're just calling something for its effects. For instance we might have a parrot that we like to encourage to say silly things, maybe on the screen or even out loud, but we don't care much about the result. In that case we can use `<-np` which sends a message but with "no promise":

```
> (define parrot
    (a-vat 'spawn
           (lambda (bcom)
             (lambda (phrase)
               ; Since we're using displayln, we're printing this
phrase
               ; rather than returning it as a string
               (displayln (format "<parrot>: ~a...
*SQWAK!*" phrase))))))
> (b-vat 'run
         (lambda ()
           (<-np parrot "Polly wants a chiptune")))
<parrot>: Polly wants a chiptune... *SQWAK!*
```

When we don't need a promise, `<-np` is an optimization that saves us from some promise overhead. But in most of our code, `<-` performs the more common case of returning a promise.

Anyway, we should have enough information to make a better constructor for friends who are far away. Recall our definition of `^calls-friend`:

```
(define (^calls-friend bcom our-name)
  (lambda (friend)
    (define what-my-friend-said
      ($ friend our-name))
    (displayln (format "<~a>: I called my friend, and they said:"
                       our-name))
    (displayln (format "   \"~a\"" what-my-friend-said))))
```

.. we'll make a few changes and name our constructor =^messages-friend=:

```
(define (^messages-friend bcom our-name)
  (lambda (friend)
    (on (<- friend our-name)
        (lambda (what-my-friend-said)
          (displayln (format "<~a>: I messaged my friend, and they said:"
                             our-name))
          (displayln (format "   \"~a\"" what-my-friend-said)))
        #:catch
        (lambda (err)
          (displayln
           "I messaged my friend but they broke their response promise...")))))
```

(We even made it a bit more robust than our previous implementation by handling errors!)

Now we can make a version of Bob that can do a better job of holding a conversation with his far-away friend Alice:

```
> (define bob2
    (b-vat 'spawn ^messages-friend "Bob"))
> (b-vat 'call bob2 alice)
<Bob>: I messaged my friend, and they said:
   "Hello Bob, my name is Alice!"
```

Much better!

2.3.2 Making and resolving our own promises

So we know that `<-` can make promises, but it turns out we can make promises ourselves:

```
> (a-vat 'run spawn-promise-cons)
'(#<live-refr promised> . #<live-refr ^resolver>)
```

As we can see, promises come in pairs: the promise object, which we can listen to with `on`, and the resolver object, which lets us fulfill or break a promise.

We can also use `spawn-promise-values` to spawn a promise (TODO: add footnote about why we didn't earlier, multiple value return not being allowed from actors currently), which returns multiple values (which we can bind with `define-values`). We can then try resolving a promise with `on`... but of course we'll need to fulfill or break it to see anything.

```
> (a-vat 'run
         (lambda ()
           (define-values (foo-vow foo-resolver)
             (spawn-promise-values))
           (define-values (bar-vow bar-resolver)
             (spawn-promise-values))
           (define (declare-resolved result)
             (printf "Resolved: ~a\n" result))
           (define (declare-broken err)
             (printf "Broken: ~a\n" err))
           (on foo-vow
               declare-resolved
               #:catch declare-broken)
           (on bar-vow
               declare-resolved
               #:catch declare-broken)
           ($ foo-resolver 'fulfill 'yeah-foo)
           ($ bar-resolver 'break 'oh-no-bar)))
Resolved: yeah-foo
Broken: oh-no-bar
```

By the way, you may notice that there's a naming convention in Goblins (borrowed from E) to append a `-vow` suffix if something is a promise (or should be treated as one). That's a good practice for you to adopt, too.

2.3.3 Finally we have #:finally

Maybe we'd like to run something once a promise resolves, regardless of whether or not it succeds or fails. In such a case we can use the `#:finally` keyword:

```
> (a-vat 'run
         (lambda ()
           (define resolves-ok
              (spawn (lambda (bcom)
                       (lambda ()
                         "This is fine!"))))
            (define errors-out
              (spawn (lambda (bcom)
                       (lambda ()
                         (error "I am error!")))))
            (define (handle-it from-name vow)
              (on vow
                  (lambda (val)
                    (displayln
                     (format "Got from ~a: ~a" from-name val)))
                  #:catch
                  (lambda (err)
                    (displayln
                     (format "Error from ~a: ~a" from-name err)))
                  #:finally
                  (lambda ()
                    (displayln
                     (format "Done handling ~a." from-name)))))
            (handle-it 'resolves-ok (<- resolves-ok))
            (handle-it 'errors-out (<- errors-out))))
Got from resolves-ok: This is fine!
Done handling resolves-ok.
Error from errors-out: #(struct:exn:fail I am error!
#<continuation-mark-set>)
Done handling errors-out.
;; === While attempting to send message: ===
I am error!
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

2.3.4 The on-fulfilled handler of "on" is optional

Maybe all you care about is the `#:catch` or `#:finally` clause. The `on-fulfilled` argument (ie, the first positional argument) to `on` is actually optional:

```
> (define ((^throws-error bcom))
    (error "oh no"))
> (define throws-error
    (a-vat 'spawn ^throws-error))
```

```
> (a-vat 'run
         (lambda ()
           (on (<- throws-error)
               #:catch
               (lambda (err)
                 (displayln (format "The error is: ~a" err))))))
The error is: #(struct:exn:fail oh no #<continuation-mark-set>)
;; === While attempting to send message: ===
oh no
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

(Side note, this is the first time we've seen the procedure definition style used in `^throws-error`... it's a way in Racket of making a procedure that defines a procedure. Handy in Goblins! If you're new to that, these two definitions are equivalent:)

```
(define (^friend bcom my-name)
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))

(define ((^friend bcom my-name) your-name)
  (format "Hello ~a, my name is ~a!" your-name my-name))
```

2.3.5 "on" with non-promise values

`on` works just fine if you pass in a non-promise value. It'll just treat that value as if it were a promise that had resolved immediately. For example:

```
> (a-vat 'run
         (lambda ()
           (on 5
               (lambda (v)
                 (displayln (format "Got: ~a" v))))))
Got: 5
```

2.3.6 "on" can return promises too

It turns out that `on` can also return a promise! However, this isn't a very common use case, so you have to ask for it via the `#:promise?` keyword.

```
> (define ((^bakery bcom name) carb)
    (format "~a's signature ~a baking" name carb))
> (define petite-oven-bakery
    (a-vat 'spawn ^bakery "The Petite Oven"))
> (a-vat 'run
         (lambda ()
           (define smell-vow
             (on (<- petite-oven-bakery "croissants")
                 (lambda (what-you-smell)
                   (format "You smell ~a.  Heavenly!!"
                           what-you-smell))
                 #:promise? #t))
           (on smell-vow displayln)))
You smell The Petite Oven's signature croissants
baking.  Heavenly!!
```

The choice of whether or not to include `#:catch` in an `on` with `#:promise?` affects whether or how an error will propagate (or be cleaned up).

Without catching an error:

```
> (define ((^throws-error bcom))
    (error "oh no"))
> (define throws-error
    (a-vat 'spawn ^throws-error))
> (a-vat 'run
         (lambda ()
           (on (on (<- throws-error)
                   (lambda (val)
                     (displayln "I won't run..."))
                   #:promise? #t)
               #:catch
               (lambda (e)
                 (displayln (format "Caught: ~a" e))))))
Caught: #(struct:exn:fail oh no #<continuation-mark-set>)
;; === While attempting to send message: ===
oh no
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

However if we catch the error, we can return a value that will succeed if we like:

```
> (a-vat 'run
         (lambda ()
           (on (on (<- throws-error)
                   #:catch
                   (lambda (e)
                     "An error?  Psh... this is fine.")
                   #:promise? #t)
               (lambda (val)
                 (displayln (format "Got: ~a" val))))))
Got: An error?  Psh... this is fine.
;; === While attempting to send message: ===
oh no
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

But if our `#:catch` handler raises an error, that error will simply propagate (instead of the original one):

```
> (a-vat 'run
         (lambda ()
           (on (on (<- throws-error)
                   #:catch
                   (lambda (e)
                     (error "AAAAH!  This is NOT fine!"))
                   #:promise? #t)
               #:catch
               (lambda (e)
                 (displayln (format "Caught: ~a" e))))))
Caught: #(struct:exn:fail AAAAH!  This is NOT fine!
#<continuation-mark-set>)
;; === While attempting to send message: ===
oh no
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
;; === While attempting to send message: ===
AAAAH!  This is NOT fine!
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:919:11
```

```
    /home/cwebber/devel/goblins/goblins/core.rkt:524:5
    /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
    /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
    /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

2.3.7 Promise pipelining

"Machines grow faster and memories grow larger. But the speed of light is constant and New York is not getting any closer to Tokyo."

from Robust Composition : Towards a Unified Approach to Access Control and Concurrency Control by Mark S. Miller

Let's say we have a car factory that makes cars:

```
(define (^car-factory bcom company-name)
  (define ((^car bcom model color))
    (format "*Vroom vroom!*  You drive your ~a ~a ~a!"
            color company-name model))
  (define (make-car model color)
    (spawn ^car model color))
  make-car)

(define fork-motors
  (a-vat 'spawn ^car-factory "Fork"))
```

Now observe... in this scenario, Fork Motors exists on `a-vat`. It will also generate a car that technically lives on `a-vat`. Let's say we live on `b-vat`... we'd still like to drive our car. Doing so seems extremely ugly:

```
> (b-vat 'run
         (lambda ()
           (on (<- fork-motors "Explorist" "blue")
               (lambda (our-car)
                 (on (<- our-car)
                     displayln)))))
*Vroom vroom!*  You drive your blue Fork Explorist!
```

This is hard to follow. Maybe if we actually name some of those promises it'll be a bit easier to read:

```
> (b-vat 'run
         (lambda ()
              (define car-vow
                (<- fork-motors "Explorist" "blue"))
              (on car-vow
                  (lambda (our-car)
                    (define drive-noise-vow
                      (<- our-car))
                    (on drive-noise-vow
                        displayln)))))
 *Vroom vroom!*  You drive your blue Fork Explorist!
```

Hm, not so much better. Naming things helped us remember what each promise was for, but it seems to be the nesting of `on` handlers that's confusing.

Fortunately, Goblins supports something called "promise pipelining". We can send an instruction to drive our car... before it even rolls off the factory lot! That's right... you can send messages to promises, before they have even resolved!

```
 > (b-vat 'run
          (lambda ()
            (define car-vow
              (<- fork-motors "Explorist" "blue"))
            (define drive-noise-vow
              (<- car-vow))
            (on drive-noise-vow
                displayln)))
 *Vroom vroom!*  You drive your blue Fork Explorist!
```

Wow... that's *much* easier to read!

But readability is not the only goal of promise pipelining. Like many things in Goblins, promise pipelining comes from the E programming language. We're still working on distributed networked Goblins code, but the foundations are there to do the same thing that E does: send messages with less round trips!

Think about it this way: if `a-vat` actually lived on a different server from `b-vat`, and both servers lived halfway across the globe with the first way we implemented things:

- `b-vat` would have to first send a message to the factory on `a-vat` first asking to make a car
- then `a-vat` would have to respond, resolving the promise with the location of the new car
- then `b-vat` would have to send *another* message asking to drive the car
- then `a-vat` would have to respond, resolving yet another promise with the noise the car makes

- and finally =b-vat= can now display the car noise to the user.

With promise pipelining, this merely becomes:

- `b-vat` would send a message to the factory on `a-vat` first asking to make a car and /at the same time/ can say, "and once this car is made, I'd like to send another message to it so I can drive it."
- then =a-vat= can make the car, drive the car, and then respond with the noise the car makes
- and now =b-vat= can display the car noise to the user.

Instead of going `B => A => B => A => B`, we have reduced our work to `B => A => B` ... a significant savings in round-trips. This can really add up in distributed applications, where (as Miller's quote at the top of this subsection indicates) we may be limited in how much we can prevent network delays by physics itself.

2.3.8 Broken promise contagion

Of course, now that we know that we can pipe promises together, what happens if an error happens in the middle of the pipeline?

```
(define (^lessgood-car-factory bcom company-name)
  (define ((^car bcom model color))
    (format "*Vroom vroom!*  You drive your ~a ~a ~a!"
            color company-name model))
  (define (make-car model color)
    (error "Your car exploded on the factory floor!  Ooops!")
    (spawn ^car model color))
  make-car)

(define forked-motors
  (a-vat 'spawn ^lessgood-car-factory "Forked"))
```

The answer is that the error is simply propagated to all pending promises:

```
> (b-vat 'run
         (lambda ()
           (define car-vow
             (<- forked-motors "Exploder" "red"))
           (define drive-noise-vow
             (<- car-vow))
           (on drive-noise-vow
                  displayln
                  #:catch
                  (lambda (err)
                    (displayln (format "Caught: ~a" err))))))
Caught: #(struct:exn:fail Your car exploded on the factory
floor!  Ooops! #<continuation-mark-set>)
;; === While attempting to send message: ===
Your car exploded on the factory floor!   Ooops!
  context...:
   /home/cwebber/devel/goblins/goblins/core.rkt:524:5
   /gnu/store/8if5yqmvz48byhinw9fs5nc5cz830ll7-racket-
7.6/collects/racket/private/more-scheme.rkt:261:28
   /home/cwebber/devel/goblins/goblins/core.rkt:989:0: actormap-turn-message
   /home/cwebber/devel/goblins/goblins/vat.rkt:135:11: lp
```

2.4 Going low-level: actormaps

2.4.1 Spawning, peeking, poking, turning

So far we have dealt with vats, but there is a lower level of abstraction in Goblins which is called an actormap.

The key differences are:

- **vat**: An event loop implementation that runs continuously in its own thread. Comes pre-built with tooling to handle message passing between vats. Actually wraps an actormap.
- **actormap**: The transactional datastructure that vats use. But, useful on its own to either build your own event loop or as a synchronous one-turn-at-a-time mapping of actor references to their current implementations.

The best way to learn is by doing, so let's make an actormap now:

```
(define am (make-actormap))
```

Let's add an actor to it. Since cells are simple, let's add one of those.

```
(define lunchbox
  (actormap-spawn! am ^cell))
```

What's in the lunchbox? We could take a look with `actormap-peek`:

```
> (actormap-peek am lunchbox)
#f
```

An empty lunchbox... we'd like to eat something later, so how about we pack ourselves a nice sandwich by using `actormap-poke`:

```
> (actormap-poke! am lunchbox 'peanut-butter-and-jelly)
> (actormap-peek am lunchbox)
'peanut-butter-and-jelly
```

What if we tried to look inside the lunchbox with `actormap-poke!` instead of `actormap-peek`?

```
> (actormap-poke! am lunchbox 'bbq-tofu)
> (actormap-poke! am lunchbox)
'bbq-tofu
```

Well that worked just fine. What happens if we do both operations with `actormap-peek`?

```
> (actormap-peek am lunchbox 'chickpea-salad)
> (actormap-peek am lunchbox)
'bbq-tofu
```

What the...??? Our lunchbox didn't change!

We said that actormaps were transactional. We have gotten a hint of this with the above: `actormap-poke!` returns a value and commits any changes. `actormap-peek` returns a value and throws any changes away.

Actually, these two functions are just simple conveniences that wrap a more powerful (but cumbersome to use) tool, `actormap-turn`:

```
> (actormap-turn am lunchbox)
'bbq-tofu
#<transactormap>
'()
'()
> (actormap-turn am lunchbox 'chickpea-salad)
#<transactormap>
'()
'()
```

`actormap-turn` returns four values to its continuation: a return value (if any... Racket's REPL doesn't print out a `(void)`, which is what actually got returned in the second invocation), a "transactormap", and two lists representing messages queued for delivery. (*TODO:*

Currently two, one for local and one for remote vats... we are likely to collapse down to just one list, so update this text when that changes!)

This `#<transactormap>` is interesting... we haven't mentioned this before, but there are really two kinds of actormaps, "whactormaps" (weak-hash actormaps) and "transactormaps" (which hold a transaction which we can choose whether or not to commit). Actually our `am` is a whactormap:

```
> am
#<whactormap>
```

It's a weak hashtable which stores the current mutable state of all actors.

But we'd like to see about that transactormap... just running `actormap-turn` on its own won't update the lunchbox either:

```
> (actormap-turn am lunchbox 'chickpea-salad)
#<transactormap>
'()
'()
> (actormap-turn am lunchbox)
'bbq-tofu
#<transactormap>
'()
'()
```

We need to capture the transactormap and commit it.

```
> (define-values (val tr-am tl tr)
    (actormap-turn am lunchbox 'chickpea-salad))
> tr-am
#<transactormap>
> (transactormap-merge! tr-am)
```

Now at least our `am` actormap should reflect that we've switched out our `'bbq-tofu` sandwich for a `'chickpea-salad` one:

```
> (actormap-peek am lunchbox)
'chickpea-salad
```

Horray!

2.4.2 Time travel: snapshotting and restoring

But what does our actormap really look like? Right now it only has one thing in it, the lunchbox cell. We can see this by "snapshotting" the actormap.

```
> (snapshot-whactormap am)
'#hasheq((#<live-refr ^cell> . #<ephemeron>))
```

This returns an immutable actormap, frozen in time. We can see that it only has one pairing in it, and yep... it looks like the key is our `lunchbox` reference:

```
> lunchbox
#<live-refr ^cell>
```

So it seems our "refr" is just a "reference"... it isn't the state of the object itself, it's something that points to that state, through an indirection of the actormap.

We now know that we can snapshot an actormap, we can freeze time and come back to it. First we'll freeze a snapshot of this period of time, change what's in the lunchbox, and then snapshot it again:

```
> (define am-snapshot1
    (snapshot-whactormap am))
> (actormap-poke! am lunchbox
                  ; sloppy joe w/ lentils
                  'sloppy-jane)
> (define am-snapshot2
    (snapshot-whactormap am))
```

Now let's restore them as separate whactormaps:

```
> (define am-restored1
    (hasheq->whactormap am-snapshot1))
> (define am-restored2
    (hasheq->whactormap am-snapshot2))
```

Now it doesn't matter what we put in our lunchbox; we can always come back to life (and lunch) as it used to be.

```
> (actormap-poke! am lunchbox 'peanut-butter-and-pickles)
> (actormap-peek am lunchbox)
'peanut-butter-and-pickles
> (actormap-peek am-restored2 lunchbox)
'sloppy-jane
> (actormap-peek am-restored1 lunchbox)
'chickpea-salad
```

Why eat leftovers when you can simply travel back in time and eat yesterday's lunch?

2.5 Advanced object patterns

2.5.1 Extending our methods

2.5.2 Mixins

2.5.3 Behavior pivoting as a kind of state machine

2.6 How this relates to "object capability security"

2.7 Sealers/unsealers and rights amplification

2.8 Networked object interactions

2.9 A peek into Goblins' machinery

2.9.1 Actors really are what they say they are

Now at last we will pull the curtain aside and see that the (hu?)man standing behind the curtain is who we thought it was all along. (This particular section is totally "optional", but hopefully illuminates some of the guts of Goblins itself.)

Here is a suitable test subject:

```
(define (^the-wizard bcom)
  ;; High and mighty
  (define the-wizard-behind-the-curtain
    (methods
     [(request-gift what)
      "HOW DARE YOU APPROACH THE ALL POWERFUL WIZARD?"]
     [(pull-back-curtain)
      (bcom the-man
            "Er... I can explain...")]))
  ;; Just a person after all
  (define the-man
    (methods
     [(request-gift what)
      (match what
        ['heart
         "Here's a clock shaped like a heart!"]
        ['brain
         "Here's a diploma!"]
```

```
          ['courage
           "The courage was inside you all along!"]
          [anything-else
           "I don't know what that is..."])]
       [(pull-back-curtain)
        "... you already pulled it back."]))
    ;; we start out obscured
    the-wizard-behind-the-curtain)

  (define wizard
    (actormap-spawn! am ^the-wizard))
```

We humbly approach the wizard asking for the gift of a heart:

```
  > (actormap-peek am wizard 'request-gift 'heart)
  "HOW DARE YOU APPROACH THE ALL POWERFUL WIZARD?"
```

Is that so? Let's snapshot time both before and after we pull back the curtain so we can remember how he changes his tone:

```
  > (define am-wizard-snapshot1
      (snapshot-whactormap am))
  > (actormap-poke! am wizard 'pull-back-curtain)
  "Er... I can explain..."
  > (define am-wizard-snapshot2
      (snapshot-whactormap am))
```

Of course now, if we request a gift of the wizard, the conversation is a bit different:

```
  > (actormap-peek am wizard 'request-gift 'heart)
  "Here's a clock shaped like a heart!"
  > (actormap-peek am wizard 'request-gift 'brain)
  "Here's a diploma!"
  > (actormap-peek am wizard 'request-gift 'courage)
  "The courage was inside you all along!"
```

Of course, this isn't too much of a surprise, since, well, we wrote the code for the wizard.

Both actormaps show our new wizard resident (and the old lunchbox cell):

```
  > am-wizard-snapshot1
  '#hasheq((#<live-refr ^cell> . #<ephemeron>)
           (#<live-refr ^the-wizard> . #<ephemeron>))
  > am-wizard-snapshot2
```

```
'#hasheq((#<live-refr ^cell> . #<ephemeron>)
         (#<live-refr ^the-wizard> . #<ephemeron>))
```

Referencing the wizard from either hashtable gives us this weird-looking "ephemeron" thing:

```
> (hash-ref am-wizard-snapshot1 wizard)
#<ephemeron>
```

Well, an ephemeron is just a datastructure to help the garbage collection in the weak hashtable of whactormaps work right. So we can peel that off:

```
> (ephemeron-value
   (hash-ref am-wizard-snapshot1 wizard))
#<mactor:local-actor>
```

What's a... "mactor"? Well, it stands for "meta-actor"... there are a few of these kinds of things. A `mactor:local-actor` is one kind of mactor (actually the most common kind), one that works in the usual way of wrapping a procedure.

It would be wonderful to be able to look at that procedure. Unfortunately, we haven't pulled in all the tools to do it yet, but there is a way to do so:

```
> (require (submod goblins/core mactor-extra))
```

This gives us access to `mactor:local-actor-handler`, which is what we want.

```
> (mactor:local-actor-handler
   (ephemeron-value
    (hash-ref am-wizard-snapshot1 wizard)))
#<procedure:methods>
```

Let's give names to both snapshotted wizard handlers:

```
> (define wizard-handler1
    (mactor:local-actor-handler
     (ephemeron-value
      (hash-ref am-wizard-snapshot1 wizard))))
> (define wizard-handler2
    (mactor:local-actor-handler
     (ephemeron-value
      (hash-ref am-wizard-snapshot2 wizard))))
```

Since these are just procedures, we can try calling them directly. As we can see, the first wizard is petulant in response to our request for a gift, whereas the second wizard, having had the curtain pulled away, is happy to assist.

```
> (wizard-handler1 'request-gift 'heart)
"HOW DARE YOU APPROACH THE ALL POWERFUL WIZARD?"
> (wizard-handler2 'request-gift 'heart)
"Here's a clock shaped like a heart!"
```

So this really was the procedure we thought it was!

But wait... if we remember correctly, our first wizard version, `the-wizard-behind-the-curtain`, returned its request to "become" something (as well as its return value) wrapped by its =bcom= capability when we called the `'pull-back-curtain` method. So what happens if we try calling that again on the unwrapped procedure?

```
> (wizard-handler1 'pull-back-curtain)
#<become>
```

That's interesting... it returned a "become" object! How can we unwrap it?

Well, here's the secret about `bcom`: as we said, it is a capability, but it's actually using a "sealer" (we'll get to what those are later, which as it sounds like is a kind of capability to "seal" an object in a certain way). To unseal it, we need the corresponding "unsealer". `mactor:local-actor` has a way of detecting that the value is wrapped in this sealer, as well as a way of unsealing it:

```
> (define wizard-become?
    (mactor:local-actor-become?
     (ephemeron-value
      (hash-ref am-wizard-snapshot1 wizard))))
> (define wizard-become-unsealer
    (mactor:local-actor-become-unsealer
     (ephemeron-value
      (hash-ref am-wizard-snapshot1 wizard))))
```

Now we can see that the value returned from the ='pull-back-curtain= method is from the wizard's `bcom` sealer:

```
> (wizard-become? (wizard-handler1 'pull-back-curtain))
#t
```

And now, at last! We can peer into its contents:

```
> (wizard-become-unsealer (wizard-handler1 'pull-back-curtain))
#<procedure:methods>
"Er... I can explain..."
```

Aha! It looks like it returns two values to its continuation... the procedure that the actor would like to become, as well as an extra (and optional, defaulting to (void)) return value. Let's bind the first of those to a useful name:

```
> (define-values (new-wizard-handler wizval)
    (wizard-become-unsealer (wizard-handler1 'pull-back-curtain)))
```

Now as you've probably guessed, this ought to be the same procedure as `wizard-handler2`. Is it?

```
; Works just like wizard-handler2...
> (new-wizard-handler 'request-gift 'courage)
"The courage was inside you all along!"
; ... because it *is* wizard-handler2!
> (eq? new-wizard-handler wizard-handler2)
#t
```

Woohoo!

Of course, the tools we've seen in this subsection are not really tools that we expect users of Goblins to use very regularly. (Indeed, when we implement better module-level security, rarely will they have access to them.) But now we've gotten a peek inside of Goblins' machinery, and seen that it was mostly what we expected all along.

I feel like all that learning is was worth a reward, don't you?

```
> (new-wizard-handler 'request-gift 'brain)
"Here's a diploma!"
```

Yes, we deserve it!

2.9.2 So how does message passing work

2.9.3 How can two actormaps communicate? (How do vats do it?)

3 Goblins API

`(require goblins)` package: `goblins`

3.1 Actors

3.1.1 What is an "actor" in Goblins?

Goblins implements the actor model on top of Racket / scheme.[2]

The fundamental operations of actors[3] are:

- An actor can send messages to other actors of which it has the address. (In Goblins, `<-`, possibly arguably `$` as well.)
- An actor may create new actors. (In Goblins, `spawn`.)
- An actor may designate its behavior for the next message it handles. (In Goblins, this means returning a new message handler by wrapping that message handler in the actor's bcom capability.)

Goblins' extensions to these ideas are:

- *Both* synchronous and asynchronous calls are defined and supported, between `$` and `<-` respectively. However, `$` is both convenient and important for systems that much be transactionally atomic (eg implementing money), it is limited to objects that are within the same vat. `<-` is more universal in that any actor on any vat may communicate with each other, but is asynchronous and cannot immediately return a resolved value.[4]
- Raw message passing without a clear way to get values back can be painful. For this reason, `<-` implicitly returns a promise that can be resolved with `on` (and, for extra convenience and fewer round trips, supports §2.3.7 "Promise pipelining").[5]

[2]Sussman and Steele famously came to the conclusion that there was in fact no difference between actor-style message passing and procedure application in the lambda calculus, and indeed both are similar. However, there is a significant difference between synchronous call-and-return procedure application (which is what scheme implements in its most general form, and between actors in Goblins is handled by `$`) and asynchronous message passing (which in Goblins is handled with `<-`).

[3]Emphasis on "fundamental" operations. Extensions happen from here and vary widely between different systems that call themselves "actors".

[4]For more on why this is, see chapters 13-15 of Mark Miller's dissertation.. This was very influential on Goblins' design, including the decision to move from a coroutine-centric approach to an E-style promise approach. It could be that coroutines are re-added, but would have to be done with extreme care; section 18.2 of that same thesis for an explaination of the challenges and a possible solution for introducing coroutines.

[5]In this sense, `<-np`, which does not return a promise, is closer to the foundational actors message passing. The kind of value that promises give us can be constructed manually by providing return addresses to `<-np`, but this is painful given how common needing to operate on the result of an operation is.

- Goblins composes with all the existing machinery of Racket/scheme, including normal procedure calls. Instead, Goblins builds its abstractions on top of it, but none of this needs to be thrown away.[6]

3.1.2 Constructors and bcom

A *constructor* is a procedure which builds the first message handler an actor will use to process messages / invocations. The constructor has one mandatory argument, traditionally called bcom (pronounced "become" or "bee-com") which can be used to set up a new message handler for future invocations.

```
> (require goblins)
> (define am (make-actormap))
; Outer procedure is the constructor.
; Implicitly takes a bcom argument.
; count argument may or may not be supplied by spawner.
> (define (^noisy-incrementer bcom [count 0])
    ; Our message handler.
    (lambda ([increment-by 1])
      (let ([new-count (+ count increment-by)])
        ; Here we create a new version of ^noisy-incrementer
        ; with count scoped to a new incremented version.
        ; The second argument to bcom specifies a return value,
        ; would return void if unspecified.
        (bcom (^noisy-incrementer bcom new-count)
              (format "My new count is: ~a" new-count)))))
> (define incr1
    (actormap-spawn! am ^noisy-incrementer))
> (actormap-poke! am incr1)
"My new count is: 1"
> (actormap-poke! am incr1 20)
"My new count is: 21"
> (define incr2
    (actormap-spawn! am ^noisy-incrementer 18))
> (actormap-poke! am incr2 42)
"My new count is: 60"
```

bcom, as shown above, is a capability (or technically a "sealer") to become another object. However, bcom does not apply a side effect; instead, it wraps the procedure and must be returned from the actor handler to set that to be its new message handler. Since this clobbers the space we would normally use to return a value (for whatever is waiting on the other end

[6]Scheme is a beautiful language to build Goblins on top of, but actually we could build a Goblins-like abstraction layer on top of any programming language with sane lexical scoping and weak hash tables (so that actors which are no longer referenced can be garbage collected).

of a `$` or a promise), bcom supports an optional second argument, which is that return value. If not provided, this defaults to `(void)`.

3.2 Core procedures

The following procedures are the core API used to write Goblins code. All of them must be run within an "actor context", which is to say either from an actor running within a vat or an actormap or within one of the procedures used to bootstrap the vat/actormap.

```
(spawn constructor argument ...) → live-refr?
  constructor : procedure?
  argument : any/c
```

Spawn an actor built by *`constructor`* with the rest of the *`argument`*s being passed to that constructor. Returns a live reference to the newly spawned actor.

The *`constructor`* is, as the name sounds, a goblins constructor, and is first passed a `bcom` argument (which is a way specify how it will behave on its next invocation) and then is passed the remaining *`argument`*s.

```
($ actor-refr arg ...) → any/c
  actor-refr : near-refr?
  arg : any/c
```

Pronounced "call", "dollar call", or "money-call".[7]

Provide a synchronous call against the current message handler of *`actor-refr`*, which must be a near `live-refr?` in the same vat as the currently running actor context. The value returned is that which is returned by the *`actor-refr`*'s message handler upon invocation (or, if `actor-ref` chose to `bcom` something else, the second argument passed to its `bcom`, or void if not provided.) Exceptions raised by *`actor-refr`*'s invocation will be propagated and can be captured.

Note that excape continuations can be set up between a caller of `$` and can be used by the callee. However, a continuation barrier is installed and so capturing of the continuation is not possible. (Note that in the future this may be relaxed so that async/await coroutines can be used with mutual consent of caller/callee; whether or not this is a good idea is up for debate.)

```
(<- actor-refr arg ...) → [promise live-refr?]
  actor-refr : live-refr?
  arg : any/c
```

[7] Why "money call"? Because you need `$` to make distributed ocap financial instruments!

Like $, but called asynchronously and returns a promise, which may be handled with on. Unlike $, <- is not limited to only near *actor-refr*s; interaction with far actors are permitted as well.

```
(<-np actor-refr arg ...) → void?
  actor-refr : live-refr?
  arg : any/c
```

Like <- but does not return (or require the overhead of) a promise. <-np is effectively an optimization; where promises are not needed, using <-np over <- is a good idea.

```
(on  vow
    [on-fulfilled
     #:catch on-broken
     #:finally on-finally
     #:promise? promise?]) → (or/c live-refr? void?)
  vow : (or/c live-refr? any/c)
  on-fulfilled : (or/c procedure? refr? #f) = #f
  on-broken : (or/c procedure? refr? #f) = #f
  on-finally : (or/c procedure? refr? #f) = #f
  promise? : bool? = #f
```

Sets up promise handlers for *vow*, which is typically a live-refr? to a promise but may be anything.

on-fulfilled, *on-broken*, and *on-finally* all, if specified, may be either a procedure (the most common case) which is run when *vow* becomes resolved, or a reference to an actor that should be messaged when the promise is resolved with the same arguments that would be passed to an equivalent procedure. As their names suggest, *on-fulfilled* will run if a promise is fulfilled and takes one argument, the fulfilled value. *on-broken* will run if a promise is broken and takes one argument, the error value. *on-finally* will run when a promise is resolved, no matter the outcome and is called with no arguments.

If #:promise? is set to #t, then on will itself return a promise. The promise will be resolved as follows:

- If *vow* is fulfilled and *on-fulfilled* is not provided, the promise returned will share *vow*'s resolution.
- If *vow* is broken and *on-broken* is not provided, the promise returned will share *vow*'s broken result.
- If *vow* is fulfilled and *on-fulfilled* is provided, the resolution will be whatever is returned from *on-fulfilled*, unless *on-fulfilled* raises an error, in which case the promise will be broken with its error-value set to this exception.

- If *vow* is broken and *on-broken* is provided, the resolution will be whatever is returned from *on-broken*, unless *on-broken* raises an error, in which case the promise will be broken with its error-value set to this exception (which may even be the original exception).

3.3 References

A *reference* is a capability to communicate with an actor. References are an indirection and abstractly correspond to an actor handler in an actormap somewhere, often in a vat.

```
(refr? obj) → bool?
  obj : any/c
```

Returns `#t` if *obj* is a reference.

3.3.1 Live vs Sturdy references

The most common kind of reference is a *live* reference, meaning that they correspond to some actor which we have an established connection to. However some references may be *sturdy* references, meaning they are serialized in such a way that they probably refer to some actor, but the connection to it is dormant in this reference itself. A sturdy reference must be enlivened with `enliven` before it can be used. (**TODO:** or do we just want to reuse `<-`? Dunno.)

NOTE: Sturdy references aren't implemented yet, and neither is `enliven`. Change that!

```
(live-refr? obj) → bool?
  obj : any/c
```

Returns `#t` if *obj* is a live reference.

```
(sturdy-refr? obj) → bool?
  obj : any/c
```

Returns `#t` if *obj* is a sturdy reference.

TODO: Define and document `enliven`, maybe.

3.3.2 Near vs far references

An actor is *near* if it is in the same vat as the actor being called in an actor context. An actor is *far* if it is not. The significance here is that only near actors may perform immediate calls with `$`, whereas any actor may perform asynchronous message sends with `<-` and `<-np`.

```
(near-refr? obj) → bool?
  obj : any/c
```

Returns `#t` if *`obj`* is a near reference.

3.3.3 Local vs remote references

Well, once machines exist, this will matter, but they don't yet :P

3.4 Actormaps

An *actormap* is the key abstraction that maps actor references to their current method handlers. There are actually two kinds of actormaps, whactormaps and transactormaps.

```
(make-actormap [#:vat-connector vat-connector]) → whactormap?
  vat-connector : (or/c procedure? #f) = #f
```

Alias for `make-whactormap`, since this is the most common actormap users make.

Actormaps are also wrapped by vats. More commonly, users will use vats than actormaps directly; however, there are some powerful aspects to doing so, namely for strictly-synchronous programs (such as games) or for snapshotting actormaps for time-traveling purposese.

In general, there are really two key operations for operating on actormaps. The first is `actormap-spawn`, which is really just used to bootstrap an actormap with some interesting actors. Actors then operate on *turns*, which are basically a top-level invocation; the core operation for that is `actormap-turn`. This can be thought of as like a toplevel invocation of a procedure at a REPL: the procedure called may create other objects, instantiate and call other procedures, etc, but (unless some portion of computation goes into an infinite loop) will eventually return to the REPL with some value. Actormap turns are similar; actors may do anything that actors can normally do within the turn, including spawning new actors and calling other actors, but the turn should ideally end in some result (as well as some new messages to possibly dispatch).[8]

3.4.1 Actormap methods

[8]Due to the halting problem, this cannot be pre-guaranteed in a turing-complete environment such as what Goblins runs in. Actors can indeed go into an infinite loop; in general the security model of Goblins is to assume that actors in the same vat can thus "hose their vat" (but really this means, an actormap turn might not end on its own, and vats currently don't try to stop it). Pre-emption can be layered manually though when operating on the actormap directly; if you want to do this, see §??? "[missing]".

```
(actormap? obj) → bool?
  obj : any/c
```

Determines if *obj* is an actormap.

```
(actormap-spawn actormap constructor arg ...)
 → [actor-refr live-refr?]
   [new-actormap transactormap?]
  actormap : actormap?
  constructor : procedure?
  arg : any/c
```

Like `spawn`, but low-level and transactional. Returns two values to its continuation, the new actor live reference, and a transactormap representing the change.

```
(actormap-spawn! actormap
                 constructor
                 arg ...)    → [actor-refr live-refr?]
  actormap : actormap?
  constructor : procedure?
  arg : any/c
```

Like `actormap-spawn!`, but directly commits the actor to *actormap*. Only returns the tech{reference} of the new actor. No changes are committed in an exceptional condition.

```
(actormap-turn actormap to-refr args ...)
 → [result any/c]
   [new-actormap transactormap?]
   [to-near (listof message?)]
   [to-far (listof message?)]
  actormap : actormap?
  to-refr : live-refr?
  args : any/c
```

Similar to performing `$`, applying *args* to *to-refr*, but transactional and a little bit cumbersome to use. (In many cases, you'll prefer to use `actormap-peek`, `actormap-poke!`, `actormap-run`, or `actormap-run!` which are easier.) Returns four values to its continuation: the result of applying *args* to *to-refr*, a transactional new actormap, and two lists of messages that may need to be sent (one to near actors, one to far actors).

```
(actormap-reckless-poke! actormap
                         to-refr
                         arg ...) → [actor-refr live-refr?]
  actormap : whactormap?
  to-refr : live-refr?
  arg : any/c
```

Like actormap-poke!, but only usable on a whactormap and mutates all bcom-effects immediately to the mapping. A little bit faster but non-transactional... corrupt state can occur in the case of exceptional conditions, as the system will not "roll back". *Use with caution!*

```
(actormap-poke! actormap to-refr args ...) → any/c
  actormap : actormap?
  to-refr : live-refr?
  args : any/c
```

Similar to performing $, applying *args* to *to-refr*. Commits its result immediately, barring an exceptional condition.

```
(actormap-peek actormap to-refr args ...) → void?
  actormap : actormap?
  to-refr : live-refr?
  args : any/c
```

Like actormap-poke!, but does not commit its result. Useful for interrogating an actor in an actormap without allowing for become-effects within it.

```
(actormap-run actormap proc) → any/c
  actormap : actormap?
  proc : (-> any/c)
```

Run *proc*, which is a thunk (procedure with no arguments) in the actormap context, but do not commit its results, instead returning its value.

Like actormap-peek, this is useful for interrogating an actormap, but can be useful for doing several things at once.

```
(actormap-run! actormap proc) → any/c
  actormap : actormap?
  proc : (-> any/c)
```

Like actormap-run but, barring exceptional conditions, does commit its results.

3.4.2 whactormap

A *whactormap* is the default kind of actormap; uses a weak hashtable for mapping.

```
(make-whactormap [#:vat-connector vat-connector]) → whactormap?
  vat-connector : (or/c procedure? #f) = #f
```

Makes a weak hashtable actormap. Used to mutably track the current state of actors.

```
(whactormap? obj) → bool?
  obj : any/c
```

Determines if `obj` is a whactormap.

3.4.3 transactormap

A *transactormap* is an actormap that stores a delta of its changes and points at a previous actormap. It must be committed using `transactormap-commit!` before its changes officially make it into its parent.

```
(transactormap? obj) → bool?
  obj : any/c
```

Returns `#t` if `obj` is a transactormap.

```
(make-transactormap parent
                    [#:vat-connector vat-connector])
 → transactormap?
  parent : actormap?
  vat-connector : (or/c procedure? #f) = #f
```

Makes a new transactormap which is not yet committed and does not have any new changes. It is unlikely you will need this procedure, since `actormap-turn`, `actormap-spawn` and friends produce it for you.

```
(transactormap-merge! transactormap) → void/c
  transactormap : transactormap?
```

Recursively merges this and any parent transactormaps until it reaches the root whactormap.

Note that creating two forking timelines of transactormaps upon a whactormap and merging them may corrupt your whactormap.

```
(transactormap-merged? transactormap) → bool?
  transactormap : transactormap?
```

Returns `#t` if this transactormap has been merged.

```
(transactormap-parent transactormap) → actormap?
  transactormap : transactormap?
```

Returns the parent actormap of this transactormap.

```
(transactormap-delta transactormap) → hasheq?
  transactormap : transactormap?
```

Returns the delta of changes to this transactormap. Mutating this yourself is not prevented but is highly inadvisable.

3.4.4 Snapshotting and restoring actormaps

```
(snapshot-whactormap whactormap) → hasheq?
  whactormap : whactormap?
```

Snapshots a whactormap by transforming it into a `hasheq?` table mapping references to ephemeron wrapped actor handlers.

```
(hasheq->whactormap ht) → whactormap?
  ht : hasheq?
```

Restores a whactormap from the *`ht`* snapshot.

3.4.5 Extra actormap procedures

```
(require (submod goblins/core actormap-extra))
```

These are very low level but can be useful for interrogating an actormap.

TODO: document.

3.4.6 Vat connectors

Somewhat awkwardly named since they most visibly show up in actormaps, a *vat connector* is a procedure (or `#f`) which is attached to an actormap. It serves two purposes:

- To tell whether or not two actor references are near each other. If both share the same vat connector, then they are considered near.[9]
- In case actors are not near each other, this specifies how to reach the actor. The procedure is called to communicate with the remote actormap, probably by dropping a message into the queue of its vat event loop.

[9]There is one case in which this could be misleading: if both references are spawned in different actormaps that have no vat connector (ie, it is `#f`), then they likely won't appear in each others' vats.

If you are using `make-actormap`, this defaults to `#f`, meaning that all other actors that also have no vat connector will assume they are likewise near. This of course also means that an actor which *is* in a vat will have no way of communicate with an actor which isn't.

On the other hand, vats built with `make-vat` set up their own vat connectors for you.

3.5 Vats

A *vat*[10] is an event loop that wraps an actormap. In most cases, users will use vats rather than the more low-level actormaps.

Actors in vats can communicate with actors in other vats on the same machine over a vat connector. For inter-machine communication, see machines. Nonetheless, for the most part users don't need to worry about this as most inter-vat communication happens using `<-`.

```
(make-vat) → procedure?
```

Starts up a vat event loop.

Returns a procedure that can be invoked to communicate with the vat (documented below).

The returned procedure uses symbol-based method dispatch.

The procedure returned from `make-vat` is called the *vat dispatcher* and is mostly used for bootstrapping or otherwise poking at a vat. Once the actors are bootstrapped in a vat they tend to do their own thing.

The vat dispatcher supports the following symbol-dispatched methods:

- `'spawn`: Behaves like `spawn` or `actormap-spawn!`.
- `'run`: Behaves like `actormap-run!`
- `'call`: Behaves like `$` or `actormap-poke!`
- `'is-running?`: Is this vat still running?
- `'halt`: Stops the next turn from happening in the vat loop. Does not terminate the current turn, but maybe it should.

[10] "Vat" might strike you as a strange name; if so, you're not alone. The term apparently refers to the musing, "How do you know if you're a brain in a vat?" Previously "vat" was called "hive" in Goblins (and its predecessors, 8sync and XUDD); it was an independent discovery of the same concept in E. Initially, Goblins stuck with "hive" because the primary author of Goblins thought it was a more descriptive term; various ocap people implored the author to not further fragment ocap vocabulary and so the term was switched. Since then, a number of readers of this documentation have complained that "vat" is confusing, and upon hearing this story have asked for the term to be switched back. Whether it's better to avoid naming fragmentation or possibly increase naming clarity is currently up for debate; your feedback welcome!

3.6 Promises

An eventual send with `<-` returns a *promise* whose resolution will happen at some future time, either being *fulfilled* or *broken*. Fulfilled promises have resolved to a value, whereas broken promises have been resolved with an error.

Every promise has a corresponding *resolver* which is messaged (often implicitly on completion of a turn)

It turns out that it is possible to make your own promises using `spawn-promise-values` or `spawn-promise-cons`, both of which return a promise / resolver pair.

```
(spawn-promise-values) → [promise live-refr?]
                         [resolver live-refr?]
```

Spawns and returns two values to its continuation: a `"promise"` and a `"resolver"`.

```
(spawn-promise-cons)
 → (cons/c [promise live-refr?] [resolver live-refr?])
```

Just like `spawn-promise-values` but returns a cons cell of the `"promise"`/`"resolver"` pair rather than returning multiple values. This requires an extra allocation (and thus destructuring on the receiving side) but can be convenient since actors can only return one value from their message handler at a time.

Promises in Goblins work closer to E than some other languages like Javascript; notable exceptions are that `on` is used rather than ".then() sausages". Likewise, having a separate resolver object also comes from E.

One major, but perhaps not very important, difference that may not be obvious is that once a promise pointed to by a reference is resolved to something, it for all intents and purposes appears to act just like that thing. If the resolution is to a near reference, it can even be immediately called with `"$"`. However, you still need to know when you can finally dollar-call such a thing, thus you still need to use `on` anyway.

3.7 Machines

A *machine* is another layer of abstraction you don't need to worry about... yet! ;)

4 Actor lib: a collection of utilities

To be written.

4.1 Cells

4.2 Methods

4.3 Facets

4.4 Revokeable caretakers

4.5 Cartoon swearing (select-$/<-)

4.6 Etc etc...